

Abductive Analysis of Administrative Policies in Rule-based Access Control

Puneet Gupta, Scott D. Stoller, and Zhongyuan Xu

Abstract—In large organizations, access control policies are managed by multiple users (administrators). An administrative policy specifies how each user in an enterprise may change the policy. Fully understanding the consequences of an administrative policy in an enterprise system can be difficult, because of the scale and complexity of the access control policy and the administrative policy, and because sequences of changes by different users may interact in unexpected ways. Administrative policy analysis helps by answering questions such as user-permission reachability, which asks whether specified users can together change the policy in a way that achieves a specified goal, namely, granting a specified permission to a specified user. This paper presents a rule-based access control policy language, a rule-based administrative policy model that controls addition and removal of facts and rules, and an abductive analysis algorithm for user-permission reachability. Abductive analysis means that the algorithm can analyze policy rules even if the facts initially in the policy (e.g., information about users) are unavailable. The algorithm does this by computing minimal sets of facts that, if present in the initial policy, imply reachability of the goal.

Index Terms—security policy, attribute-based access control, policy administration, rule-based policy, policy verification



1 INTRODUCTION

The increasingly complex security policies required by applications in large organizations are more concise and easier to administer when expressed in higher-level policy languages. Recently, frameworks with rule-based policy languages, which provide flexible support for high-level attribute-based policies, have attracted considerable attention.

In large organizations, access control policies are managed by multiple users (administrators). An *administrative framework* (also called *administrative model*) is used to express policies that specify how each user may change the access control policy. For example, several administrative frameworks have been proposed for role-based access control (RBAC) [1], starting with the classic ARBAC97 model [2].

Fully understanding the implications of an administrative policy in an enterprise system can be difficult, because of the scale and complexity of the access control policy and the administrative policy, and because sequences of changes by different users may interact in unexpected ways. Administrative policy analysis helps by answering questions such as user-permission reachability, which asks whether specified users can together change the policy in a way that achieves a specified goal, namely, granting a specified permission to a specified user. Several analysis algorithms for user-permission reachability for ARBAC97 and variants thereof have been developed, e.g., [3], [4], [5],

[6]. There is some work on administrative frameworks for rule-based access control and analysis algorithms for such frameworks [7], [8], [9], but it considers only addition and removal of facts, not rules. Analysis algorithms for ARBAC also consider, in effect, only addition and removal of facts, not rules, because the administrative operations in ARBAC correspond to addition and removal of facts.

This paper defines ACAR (“Access Control and Administration using Rules”), a rule-based access control policy language with a rule-based administrative framework that controls addition and removal of facts and rules. ACAR allows policies to be expressed concisely and at a desirable level of abstraction. Nevertheless, fully understanding the implications of an administrative policy in ACAR might be more difficult, in some ways, than fully understanding the implications of an ARBAC policy, because in addition to considering interactions between interleaved sequences of changes by different administrators, one must also consider chains of inferences using the facts and rules in each intermediate policy.

This paper presents a symbolic analysis algorithm for answering atom-reachability queries for ACAR policies, i.e., for determining whether changes by specified administrators can lead to a policy in which some instance of a specified atom (an atom is like a fact except that it may contain variables), called the goal, is derivable. To the best of our knowledge, this is the first analysis algorithm for a rule-based policy framework that considers changes to the rules in the policy as well as changes to the facts in the policy. Atom reachability can express a variety of interesting properties, including user-permission reachability.

Our algorithm translates a policy analysis problem

P. Gupta is with Google, Inc., Mountain View, CA. This work was done while he was with the Department of Computer Science, Stony Brook University, Stony Brook, NY 11794.

S. Stoller and Z. Xu are with the Department of Computer Science, Stony Brook University, Stony Brook, NY 11794. Contact email: stoller@cs.stonybrook.edu.

that involves changes to rules and facts into a problem that involves changes only to facts. We consider this approach to be a contribution of our work; we have not seen it in prior work. This approach can be adapted to other settings, but not universally. In our setting, this approach works well because it is possible to simulate a rule granting permission to add rules using one rule granting permission to add facts and one auxiliary rule, as described in Section 4.1. This is a consequence of the design of ACAR. With other administrative frameworks, simulating addition of rules using addition of facts might be difficult or inefficient.

It is often desirable to be able to analyze rule-based policies with incomplete knowledge of the facts in the initial policy; for example, a database containing those facts might not exist yet, or it might be unavailable to the policy engineer. Even if a database of facts exists and is available, more general analysis results that hold under limited assumptions about the initial facts are often preferable to results that hold for only one given set of initial facts. For example, consider the policy that a clinician at a given hospital may treat a patient if he is a member of a hospital workgroup that is treating that patient. A policy auditor might want to analyze the rules in the hospital policy to compute all sequences of administrative actions (or “plans”) that may allow a user to be a treating clinician for a patient, independent of data about specific patients, workgroups, etc. Even if such data exists and is available, it is transient, and the analysis is more thorough if it considers more general scenarios.

There are two approaches to solve such an analysis problem. In the *deductive approach*, the user specifies constraints (expressing assumptions) about the initial facts, and the analysis determines whether the goal is reachable under those constraints. However, formulating appropriate constraints might be difficult. We adopt an *abductive approach*, in which the analysis determines conditions on the set of facts in the initial policy under which the goal is reachable. More specifically, our abductive analysis determines minimal set of atoms that, if present in the initial policy, imply reachability of the goal. This approach is inspired by Becker *et al.*'s abductive policy analysis for rule-based policy languages [10], [11], and our algorithm builds on their tabling-based policy evaluation algorithm.

This paper is a revised and extended version of [12]. The major changes are replacement of the tabling algorithm in [10] with the tabling algorithm in [11] in phase 3 of our algorithm, addition of wildcards to the policy language, extension of the algorithm to produce plans, and addition of details, examples, and (in the supplemental material) correctness proofs.

2 THE ACAR FRAMEWORK

This section defines the ACAR (Access Control and Administration using Rules) framework.

t	$::= v \mid c(t^*)$	lit	$::= a_{ex} \mid a_{neg} \mid a_{in}$
a_{ex}	$::= p_{ex}(t^*)$	$rule$	$::= a_{in} :- lit^*$
a_{in}	$::= p_{in}(t^*)$	$fact$	$::= \text{ground instance}$
a_{neg}	$::= !p_{ex}((t \mid _)^*)$		$\text{of } a_{ex}$

Fig. 1. Grammar for ACAR policy language.

2.1 Policy Language and Administrative Framework

The policy language is a Prolog-like rule-based language with constructors and negation. Predicates are classified as intensional or extensional. *Intensional* predicates are defined by rules. *Extensional* predicates are defined by facts. Constructors are used to construct terms representing operations, rules (being added or removed), parameterized roles, etc. The language is parameterized by the sets of predicates, variables, and constructors. The grammar appears in Figure 1. p_{in} , p_{ex} , c , and v range over intensional predicates, extensional predicates, constructors, and variables, respectively. t , a , and lit are mnemonic for term, atom, and literal, respectively. Predicates and constructors start with a lowercase letter; variables start with an uppercase letter. Negation is denoted by “!”. Constants are represented as constructors with arity zero; the empty parentheses are elided. t^* denotes a comma-separated sequence of zero or more instances of non-terminal t . A term or atom is *ground* if it does not contain any variables. A *policy* is a set of rules and facts.

Negation and Wildcard: The grammar ensures that negation is applied only to extensional predicates. Our experience with case studies suggests that this restriction is acceptable for many typical policies. For example, in our healthcare network case study, membership in workgroups is recorded in an extensional predicate, and a negative premise involving that predicate ensures that a manager u cannot appoint the head of a workgroup of which u is a member.

To increase the expressiveness, the language allows the special symbol “_”, called *wildcard*, to be used as an argument of an extensional predicate (but not as an argument of a constructor) in a negative literal. Using a wildcard as an argument in a negative premise represents a universal quantification over the value of that argument. For example, in the policy of the substance abuse facility `gcSAF` in our healthcare network case study, a clinician treating a patient can create a new encounter with a patient by adding a fact to the `encounter` predicate. The predicate `encounter(EncID, Pat, Wkgrp, Fac, Type)` means there exists a patient encounter with unique identifier `EncID` for patient `Pat` at facility `Fac` of type `Type` and is handled by workgroup `Wkgrp`. The following rule uses negation and wildcard to ensure that the identifier for the new encounter is fresh.

```

permit(Cli, addFact(encounter(EncID, Pat,
                             Wkgrp, gcSAF, Type)))
:- memberOf(Cli, trCli(Pat, gcSAF)),
   !encounter(EncID, _, _, _, _)

```

Permissions and Administrative Operations:

The intensional predicate `permit(user, operation)` specifies permissions, including permissions for administrative operations. The administrative operations are `addRule(rule)`, `removeRule(rule)`, `addFact(aex)`, and `removeFact(aex)`. Let `AdminOp = {addRule, removeRule, addFact, removeFact}`. `addRule` and `addFact` have the same effect as `assert` in Prolog. `removeRule` and `removeFact` have the same effect as `retract` in Prolog. We use separate administrative operations for facts and rules to improve readability.

The framework defines how permissions to perform administrative operations are controlled. These permissions are expressed using the `permit` predicate but given a special interpretation, as specified below in the semantics of administrative policies.

For an operation *op*, an *op permission rule* is a rule whose conclusion has the form `permit(..., op(...))`. An *administrative permission rule* is an *op permission rule* with $op \in \text{AdminOp}$.

2.2 Representation of Role-Based Access Control

Role-based access control (RBAC) can be expressed in our framework in a straightforward way. This section describes how some core features of RBAC are modeled in the running example introduced in Section 2.3 and in the healthcare network case study described in Section 5.

Role Membership: Role membership is represented by the intensional predicate `memberOf(User, Role)`. The extensional predicate `directMemberOf(User, Role)` is the direct (i.e., not including inheritance) user-role assignment. Thus, users are assigned to roles by adding facts to the `directMemberOf` predicate. The following rule expresses that a user *u* is a member of role *R* if *u* is directly assigned to *R*.

```

memberOf(User, Role)
:- directMemberOf(User, Role)

```

 (2.1)

Role hierarchy is represented by recursive rules defining `memberOf`. For example, the rule `memberOf(User, r1) :- memberOf(User, r2)` expresses that role *r1* is senior to role *r2*.

Role Activation: A member of a role must activate the role to use the permissions granted to that role [1]. Activation of role *Role* for user *User* is expressed by adding the fact `hasAct(User, Role)` to the extensional relation `hasAct`. The following rules express that a user can activate a role of which he is a member, and that a user can deactivate any of his activated roles.

```

permit(User, addFact(hasAct(User, Role)))
:- memberOf(User, Role)

```

 (2.2)

```

permit(User, removeFact(hasAct(User, Role)))
:- hasAct(User, Role)

```

 (2.3)

2.3 Running Example

As a running example, we use a fragment of the healthcare network case study described in Section 5. The running example focuses on the policy for appointing a user as a treating clinician for a patient at `gwHosp` (“get well hospital”), a hospital in the healthcare network. The policy officer at `gwHosp` can add rules that define membership in the `trCli` role. We refer to the policy officer at `gwHosp` as the HPO, mnemonic for Hospital Policy Officer.

Predicates used in this example include `consentTT(Pat, Cli, Fac)`, which means clinician *Cli* has patient *Pat*’s consent to treat him at facility *Fac*, and `encounter(EncID, Pat, Wkgrp, Fac, Type)`, which means there is a patient encounter with unique identifier *EncID* for patient *Pat* at facility *Fac* of type *Type* and being handled by workgroup *Wkgrp*.

Roles used in this example include the following. Members of `trCli(Pat, Fac)` are treating clinicians for patient *Pat* at facility *Fac*. Members of `pOfc(Fac)` are policy officers at facility *Fac*. Members of `cli(Fac, Spcty)` are clinicians at facility *Fac* under specialty *Spcty*. Members of `wkgrp(W, Fac, Spcty, WT)` are members of the workgroup *W*, which is of type *WT* (mnemonic for “Workgroup Type”), under specialty *Spcty* at facility *Fac*. Members of `patient` are patients. Members of `agent(Pat)` are agents of patient *Pat*.

The running example policy appears in Figure 2. It allows HPO to define the `trCli` role using the following two kinds of rules: if the user has at least explicit consent to treatment for a patient, then he can be a treating clinician for that patient; if the user is at least a member of a workgroup that is treating the patient, then that user can be a treating clinician for that patient. Rules (2.4) and (2.5) allow the HPO to add these two kinds of rules, respectively. In this description, “at least” indicates that the stated requirement is the minimal one; the HPO may impose additional requirements, by including additional premises in added rules, as discussed in Section 2.4.

Rules (2.6) and (2.8) allow HPO to add rules that allow patients and their agents, respectively, to grant consent to treatment. Rules (2.7) and (2.9) allow HPO to add rules that allow patients and their agents, respectively, to revoke consent to treatment.

To help express queries, we also include in the policy a few facts about prototypical users, stating that `cli1` is a surgeon at `gwHosp`, `pat1` is a patient, and `hpo1` is a `gwHosp` policy officer. These facts appear at the bottom of Figure 2.

```

permit (User,
  addRule(memberOf(Cli, trCli(Pat, gwHosp))
    :- consentTT(Pat, Cli, gwHosp)))
:- hasAct (User, pOfc(gwHosp))      (2.4)

permit (User,
  addRule(memberOf(Cli, trCli(Pat, gwHosp))
    :- hasAct(Cli, cli(gwHosp, Spc)),
    memberOf(Cli, wkgp(W, gwHosp, Spc, WT)),
    encounter(EncID, Pat, W, gwHosp, Type)))
:- hasAct (User, pOfc(gwHosp))      (2.5)

permit (User,
  addRule(permit (Pat, addFact (consentTT (Pat,
    Cli, gwHosp)))
    :- hasAct (Pat, patient)))
:- hasAct (User, pOfc(gwHosp))      (2.6)

permit (User,
  addRule(permit (Pat, removeFact (consentTT (Pat,
    Cli, gwHosp)))
    :- hasAct (Pat, patient)))
:- hasAct (User, pOfc(gwHosp))      (2.7)

permit (User,
  addRule(permit (Ag, addFact (consentTT (Pat,
    Cli, gwHosp)))
    :- hasAct (Ag, agent (Pat))))
:- hasAct (User, pOfc(gwHosp))      (2.8)

permit (User,
  addRule(permit (Ag, removeFact (consentTT (Pat,
    Cli, gwHosp)))
    :- hasAct (Ag, agent (Pat))))
:- hasAct (User, pOfc(gwHosp))      (2.9)

hasAct (cli1, cli (gwHosp, surgeon))
hasAct (pat1, patient)
hasAct (hp01, pOfc (gwHosp))

```

Fig. 2. Running example.

2.4 Semantics

A rule is *safe* if it satisfies the following conditions. (1) Every variable that appears in the conclusion outside the arguments of `addRule` and `removeRule` also appears in a positive premise. (2) Every variable that appears in a negative premise also appears in a positive premise. (3) In every occurrence of `permit`, the second argument is a constructor term, not a variable. (4) Every occurrence of `addRule` or `removeRule` is in the second argument of `permit` in the conclusion of a rule. A policy is safe if all rules in the policy are safe. Note that condition (1) is essentially the conventional notion of safety in logic programs, which, for languages like ours that do not contain equality premises, requires that every variable that appears in the conclusion also appears in a positive premise.

A policy P is *well-formed* if (1) P is safe, (2) the argument to each occurrence of `addFact` and `removeFact` in P is an extensional atom (not necessarily ground), and (3) for each extensional predicate p , if a wildcard is used as an argument to p in any

rule in P , then P does not contain `removeFact` permission rules for p (Section 4.4 explains the reason for this requirement).

Intuitively, the semantics $\llbracket P \rrbracket$ of a policy P contains all atoms deducible from P . Formally, the semantics $\llbracket P \rrbracket$ of a policy P is the least fixed-point of F_P , defined by

$$\begin{aligned}
 F_P(I) = \{a\theta \mid & (a :- a_1, \dots, a_m, !b_1, \dots, !b_n) \in P \\
 & \wedge (\forall i \in [1..m] : a_i\theta \in I) \\
 & \wedge (\forall i \in [1..n] : b_i\theta \notin I)\}.
 \end{aligned}$$

To simplify notation, this definition assumes that the positive premises appear before the negative premises; this does not affect the semantics. We sometimes write $P \vdash a$ (read “ P derives a ”) to mean $a \in \llbracket P \rrbracket$. In the definition of F_P , if b_i contains wildcards, $b_i \notin I$ holds if I contains no terms that match b_i , where a wildcard matches any term.

Fixed Administrative Policy: Our goal in this work is to analyze a changing access control policy subject to a fixed administrative policy. Therefore, we consider policies that satisfy the *fixed administrative policy requirement*, which says that administrative permission rules cannot be added or removed, except that `addFact` administrative permission rules can be added. This exception is useful in practice and can be accommodated easily in the reachability analysis.

We formalize this requirement as follows. A *higher-order* administrative permission rule is an administrative permission rule whose conclusion has the form `permit(..., op(permit(..., op'(...)))` with $op \in \text{AdminOp}$ and $op' \in \text{AdminOp}$. A rule satisfies the fixed administrative policy requirement if either it is not a higher-order administrative permission rule or it is an administrative permission rule having the above form with $op = \text{addRule}$ and $op' = \text{addFact}$. A policy satisfies the fixed administrative policy requirement if all of the rules in it do.

Even in a policy with no higher-order administrative permission rules, the available administrative permissions may vary, because addition and removal of other rules and facts may change the truth values of the premises of administrative permission rules.

Administrative Policy Semantics: The above semantics is for a fixed policy. We specify the semantics of administrative operations and administrative permissions by defining a transition relation T between policies, such that $\langle P, u : op, P' \rangle \in T$ iff policy P permits user u to perform administrative operation op thereby changing the policy from P to P' . We refer to $u:op$ as an *administrative action*.

Rule R is *at least as strict as* rule R' if (1) R and R' have the same conclusion, and (2) the set of premises of R is a superset of the set of premises of R' . Comparison of rules ignores renaming of variables (in other words, it is based on α -equality).

$\langle P, u : \text{addRule}(R), P \cup \{R\} \rangle \in T$ if there exists a rule R' such that (1) R is at least as strict as R' , (2)

$P \vdash \text{permit}(u, \text{addRule}(R'))$, (3) $R \notin P$, (4) R satisfies the fixed administrative policy requirement, and (5) R satisfies the safe policy requirement. Note that R' may be a partially or completely instantiated version of the argument of `addRule` in the `addRule` permission rule used to satisfy condition (2); this follows from the definition of \vdash . Thus, an administrator adding a rule may specialize the “rule pattern” in the argument of `addRule` by instantiating some of the variables in it and by adding premises to it; the motivation for this is illustrated below. We call the argument of `addRule` or `removeRule` a “rule pattern”, even though it is generated by the same grammar as rules, to emphasize that it can be specialized in these ways.

$\langle P, u : \text{removeRule}(R), P \setminus \{R\} \rangle \in T$ if there exists a rule R' such that R is as least as strict as R' , $P \vdash \text{permit}(u, \text{removeRule}(R'))$, and $R \in P$.

$\langle (P, u : \text{addFact}(a), P \cup \{a\}) \in T$ if $P \vdash \text{permit}(u, \text{addFact}(a))$ and $a \notin P$.

$\langle (P, u : \text{removeFact}(a), P \setminus \{a\}) \in T$ if $P \vdash \text{permit}(u, \text{removeFact}(a))$ and $a \in P$.

Discussion of Semantics of `addRule`: Our semantics for `addRule` permission rules allows addition of rules that are stricter than the specified rule patterns. This greatly increases flexibility for administrators to customize rules being added, while not allowing them to add rules that violate desired safety properties. For example, the healthcare network’s policy might contain the following rule, which allows a facility’s policy officer to add rules allowing the facility’s human resource (HR) manager to appoint users who have federal certification for medical practice as clinicians at that facility by making them direct members of the `clinician` role.

```
permit(PO,
  addRule(permit(HR,
    addFact(directMemberOf(Cli,
      cli(Facility, Spcty))))
    :- memberOf(HR, hrManager(Facility)),
      fedCertCli(Cli)))
:- hasAct(PO, pOfc(Facility))
```

Using this administrative rule, a `gWHosp` policy officer is permitted to add a rule with additional premises that restrict the Human Resources (HR) manager to appoint only clinicians who are also certified by the state. For example, `pOfc(gWHosp)` might add the following rule to `gWHosp` policy:

```
permit(HR, addFact(directMemberOf(Cli,
  cli(Facility, Spcty))))
:- memberOf(HR, hrManager(Facility)),
  fedCertCli(Cli), stateCertCli(Cli)
```

3 ABDUCTIVE REACHABILITY

This section defines abductive atom-reachability queries, solutions to such queries, and comprehensive solutions to such queries. A solution describes one initial state from which the goal in the query is

reachable; a comprehensive solution describes all such initial states.

Let a and b denote atoms, L denote a literal, and \vec{L} denote a sequence of literals. An atom a is *subsumed* by an atom b , denoted $a \preceq b$, iff there exists a substitution θ such that $a = b\theta$. For an atom a and a set A of atoms, let $\llbracket a \rrbracket = \{a' \mid a' \preceq a\}$ and $\llbracket A \rrbracket = \bigcup_{a \in A} \llbracket a \rrbracket$.

A *specification of abducible atoms* is a pair $A = \langle Ab, nAb \rangle$, where Ab and nAb are sets of extensional atoms. Instances of atoms in Ab are abducible, except instances of atoms in nAb are not abducible. More formally, an atom a is abducible with respect to $\langle Ab, nAb \rangle$ if $a \in \llbracket \langle Ab, nAb \rangle \rrbracket$, where $\llbracket \langle Ab, nAb \rangle \rrbracket = \llbracket Ab \rrbracket \setminus \llbracket nAb \rrbracket$.

Given an initial policy P_0 , a set U_0 of users (the active administrators), and a transition relation τ on policies, the *state graph* for P_0 , U_0 , and τ , denoted $\text{SG}(P_0, U_0, \tau)$, contains policies reachable from P_0 by actions of users in U_0 according to transition relation τ . Specifically, $\text{SG}(P_0, U_0)$ is the least graph $\langle N, E \rangle$ such that (1) $P_0 \in N$ and (2) $\langle P, u : \text{op}, P' \rangle \in E$ and $P' \in N$ if $P \in N \wedge u \in U_0 \wedge \langle P, u : \text{op}, P' \rangle \in \tau$. Note that the parameter τ in this definition may be instantiated with the transition relation T defined in Section 2.4 or restricted versions of T defined later.

An *abductive atom-reachability query* is a tuple $\langle P_0, U_0, A, G_0 \rangle$, where P_0 is a policy (the initial policy), U_0 is a set of users (the users trying to reach the goal), A is a specification of abducible atoms, and G_0 is an atom called the goal. Informally, P_0 contains rules and facts that are definitely present in the initial state, and $\llbracket A \rrbracket$ contains facts that might be present in the initial state. Other facts are definitely not present in the initial state and, since we make the closed world assumption, are considered to be false.

A *ground solution* to an abductive atom-reachability query $\langle P_0, U_0, A, G_0 \rangle$ is a tuple $\langle G, \Delta, \pi \rangle$ such that G is a ground instance of G_0 , Δ is a ground subset of $\llbracket A \rrbracket$ called the *residue*, and π is a path in $\text{SG}(P_0 \cup \Delta, U_0, T)$ from P_0 to a policy P such that $P \vdash G$. Informally, a ground solution $\langle \Delta, G, \pi \rangle$ indicates that a policy P in which G holds is reachable from $P_0 \cup \Delta$ through the sequence of administrative actions by users in U_0 that appears on the edges of π . We sometimes refer to π as a *plan*.

A *minimal-residue ground solution* to a query is a ground solution $\langle G, \Delta, \pi \rangle$ such that, for all $\Delta' \subset \Delta$, there does not exist π' such that $\langle G, \Delta', \pi' \rangle$ is a ground solution to the query.

Let $\text{GndSoln}(Q)$ and $\text{MinGndSoln}(Q)$ denote the set of ground solutions and minimal-residue ground solutions, respectively, for an abductive reachability query Q .

A *tuple disequality* has the form $\langle t_1, \dots, t_n \rangle \neq \langle t'_1, \dots, t'_n \rangle$, where the t_i and t'_i are terms.

A substitution θ is *ground*, denoted $\text{ground}(\theta)$, if it maps variables to ground terms. Let GndSubst denote the set of ground substitutions.

A *comprehensive solution* to an abductive atom-

reachability query $Q = \langle P_0, U_0, A, G_0 \rangle$ is a set S of tuples of the form $\langle G, \Delta, \pi, D \rangle$, where G is an atom (not necessarily ground), Δ is a set of atoms (not necessarily ground), π is a path (i.e., an alternating sequence of policies and administrative actions, not necessarily ground, starting and ending with a policy), and D is a set (interpreted as a conjunction) of tuple disequalities over the variables in Δ and G , such that

Soundness: S represents only ground solutions to the query, i.e., $\forall \langle G, \Delta, \pi, D \rangle \in S. \forall \theta \in \text{GndSubst}. D\theta = \text{true} \Rightarrow \langle G\theta, \Delta\theta, \pi\theta \rangle \in \text{GndSoln}(Q)$.

Comprehensiveness: S represents all minimal-residue ground solutions to the query, i.e., $\forall \langle G', \Delta', \pi' \rangle \in \text{MinGndSoln}(Q). \exists \langle G, \Delta, \pi, D \rangle \in S. \exists \theta \in \text{GndSubst}. D\theta = \text{true} \wedge G' = G\theta \wedge \Delta' = \Delta\theta$.

A variety of interesting properties can be expressed as atom reachability. User-permission reachability can be expressed as atom reachability, by taking the goal to be an appropriate instance of `permit`. For role-based policies, user-role reachability can be expressed as atom reachability, by taking the goal to be an appropriate instance of `memberOf`. Atom reachability queries can specify that a permission or role should be reachable only under certain conditions, e.g., that a role is reachable only if a user associated with that role has granted consent, as in the running example below. Separation of duty properties can be expressed as atom reachability. For example, atom reachability can be used to check whether a user can be the purchasing agent and accounting agent for a transaction, by adding a rule such as `goal() :- memberOf(U, PurchasingAgent(Trans)), memberOf(U, AccountingAgent(Trans))`.

3.1 Running Example

We illustrate abductive atom-reachability queries using the running example in Section 2.3. Our sample query asks whether a clinician may be a treating clinician without having the patient’s consent to treatment. To express this, we add the following rule to the initial policy:

```
treatingWithoutConsent(Pat, Cli)
:- memberOf(Cli, trClin(Pat, gwHosp)),
   !consentToTreatment(Pat, Cli, gwHosp)
```

The initial policy P_0 contains the rules and facts in Section 2.3 and this rule. The set U_0 of active administrators is $\{\text{hpo1}, \text{pat1}\}$. The specification of abducible atoms is $\{\{\text{memberOf}(\text{User}, \text{wkgp}(W, \text{gwHosp}, \text{Spcty}, \text{WT})), \text{encounter}(\text{EncID}, \text{Pat}, W, \text{gwHosp}, \text{Type})\}, \emptyset\}$. The goal G_0 is `treatingWithoutConsent(pat1, cli1)`.

3.2 Undecidability

The abductive atom-reachability problem is undecidable. We prove this by reduction from the user-

role reachability problem for PARBAC without role hierarchy, which is known to be undecidable [6]. The reduction is straightforward and is described in Section 8 in the supplemental material.

4 ANALYSIS ALGORITHM

The algorithm has four phases. Phase 1 transforms the policy to simulate `addRule` and `removeRule` (in other words, the effects of adding and removing rules are simulated without actually adding and removing rules). Phase 2 transforms the policy to simulate `addFact` and `removeFact`. Phase 3 is a modified version of Becker *et al.*’s algorithm for tabled evaluation with abduction; it produces candidate solutions. The policy transformations and algorithm modifications are necessary because the original version of the algorithm is designed to derive a goal from a fixed policy. The transformations and modifications together enable the modified algorithm to compute sets of policy updates (i.e., administrative operations) needed to derive the goal. However, Phase 3 does not consider the order in which these administrative operations should be performed. Phase 4 checks all conditions that depend on the order in which administrative operations are performed. These conditions relate to negation, because in the absence of negation, removals are unnecessary, and additions can be done in any order consistent with the logical dependencies that the tabling algorithm already takes into account.

4.1 Phase 1: Simulate `addRule` transitions, eliminate `removeRule` transitions

This phase transforms the given policy P_0 into a policy $\text{simAddRule}(P_0)$ that is used instead of P_0 in subsequent phases of the algorithm. P_0 and $\text{simAddRule}(P_0)$ are not equivalent. Informally, the relationship between them is that $\text{simAddRule}(P_0)$ contains additional rules that simulate the effects of `addRule` transitions using `addFact` transitions; adding rules to simulate `removeRule` transitions is unnecessary, as discussed below. This relationship between P_0 and $\text{simAddRule}(P_0)$ is captured by the similarity relation. Policies P and P' are *similar*, denoted $P \simeq P'$, if P and P' contain the same rules and facts with two exceptions: (1) P contains no rules involving auxiliary predicates, and the set of rules in P' that involve auxiliary predicates is exactly the set of rules obtained by transforming the `addRule` permission rules in P using the `simAddRule` transformation; and (2) P contains no facts involving auxiliary predicates, and the set of facts in P' that involve auxiliary predicates are exactly those needed to simulate rules that have been added to P using `addRule` permission rules (a precise definition appears in Section 9 in the supplemental material). Similarity implies that policies are equivalent with respect to derivability of atoms: if P and P' are similar, then for each atom a for

a predicate other than an auxiliary predicate, $a \in \llbracket P \rrbracket$ iff $a \in \llbracket P' \rrbracket$.

The *no-addRule, no-removeRule transition relation* $T_{-aR, -rR}$ is defined the same way as the transition relation T in Section 2.4 except *addRule* transitions and *removeRule* transitions are eliminated. An atom is *reachable* in a state graph iff the state graph contains a policy in which that atom is derivable. The policy $\text{simAddRule}(P_0)$ produced by this phase is designed so that, for every policy P in the state graph $\text{SG}(P_0, U_0, T)$, the state graph $\text{SG}(\text{simAddRule}(P_0), U_0, T_{-aR, -rR})$ contains a policy similar to P . This implies the same atoms are reachable in these state graphs.

Eliminate *removeRule* transitions: To see why it is safe to simply eliminate *removeRule* transitions, without including rules that simulate them in P' , note that such transitions remove only rules defining intensional predicates, and hence the effect of such transitions is to make intensional predicates smaller. Since negation cannot be applied to intensional predicates, making intensional predicates smaller never makes more facts (including instances of the goal) derivable. Therefore, every instance of the goal that is derivable in some policy in $\text{SG}(P_0, U_0, T)$ is derivable in some policy in $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$, where the *no-removeRule transition relation* $T_{\downarrow aR, -rR}$ is defined the same way as the transition relation T in Section 2.4 except *removeRule* transitions are eliminated. Conversely, since $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ is a subgraph of $\text{SG}(P_0, U_0, T)$, every instance of the goal that is derivable in some policy in $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$, is derivable in some policy in $\text{SG}(P_0, U_0, T)$. Therefore, elimination of *removeRule* transitions does not affect the answer to abductive atom-reachability queries.

Simulate *addRule* transitions: We add rules that use *addFact* to simulate the effect of *addRule*. Specifically, the policy $\text{simAddRule}(P)$ is obtained from P as follows. Let R be an *addRule* permission rule $\text{permit}(U, \text{addRule}(L :- \vec{L}_1)) :- \vec{L}_2$ in P . Two rules are added to simulate R . One rule is the rule pattern in the argument of *addRule*, extended with an additional premise using a fresh extensional predicate aux_R that is unique to the rule: $L :- \vec{L}_1, \text{aux}_R(\vec{X})$, where the vector of variables \vec{X} is $\vec{X} = \text{vars}(L :- \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$. The other is an *addFact* permission rule that allows the user to add facts to this new predicate: $\text{permit}(U, \text{addFact}(\text{aux}_R(\vec{X}))) :- \vec{L}_2$. The auxiliary predicate aux_R keeps track of which instances of the rule pattern have been added. Recall from Section 2.1 that users are permitted to instantiate variables in the rule pattern when adding a rule. Note that users must instantiate variables that appear in the rest of the *addRule* permission rule, i.e., in $\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2)$, because if those variables are not grounded, the *permit* fact necessary to add the rule will not be

derivable using rule R . Therefore, each fact in aux_R records the values of those variables. In other words, an *addRule* transition t in $\text{SG}(P_0, U_0, T)$ in which the user adds an instance of the rule pattern in the argument of *addRule* in R with \vec{X} instantiated with \vec{c} is “simulated” in $\text{SG}(\text{simAddRule}(P_0), U_0, T_{-aR, -rR})$ by an *addFact* transition t that adds $\text{aux}_R(\vec{c})$.

$\text{SG}(P_0, U_0, T)$ also contains transitions t' that are similar to t except that the user performs additional specialization of the rule pattern by instantiating additional variables in the rule pattern or by adding premises to it. Those transitions are eliminated by this transformation, i.e., there are no corresponding transitions in $\text{SG}(\text{simAddRule}(P_0), U_0, T_{-aR, -rR})$. This is sound, because those transitions lead to policies in which the intensional predicate p that appears in literal L (i.e., L is $p(\dots)$) is smaller, and as argued above, since negation cannot be applied to intensional predicates, eliminating transitions that lead to smaller intensional predicates does not affect the answer to abductive atom-reachability queries. This is the technical meaning of the informal statement in Section 2 that allowing administrators to add stricter rules does not enable them to violate safety requirements.

Example: Figure 3 presents the rules added to P_0 by the *simAddRule* transformation for the running example introduced in Sections 2.3 and 3.1. Recall that the initial policy P_0 consists of all the rules and facts presented in those sections. Note that a nullary predicate may be empty (i.e., contain no facts) or it may contain a single fact represented by a 0-tuple.

4.2 Phase 2: Simulate *addFact* transitions and *removeFact* transitions

The transformation in this phase adds rules that use ordinary inference to simulate the effects of *addFact* and *removeFact* transitions. For example, an *addFact* permission rule that allows addition of a fact a is simulated by a rule that makes a derivable in the current policy. Similarly, an *removeFact* permission rule that allows removal of a fact a is simulated by a rule that makes $\neg a$ derivable in the current policy.

Specifically, for each *addFact* permission rule $\text{permit}(U, \text{addFact}(a)) :- \vec{L}$, the transformation adds the rule $a :- \vec{L}, \text{u0}(U)$. The transformation also introduces a new extensional predicate u0 and, for each $u \in U_0$, the fact $\text{u0}(u)$ is added to the policy. For example, to simulate rule (4.1) in Figure 3, the transformation adds the rule:

```
consentTT(Pat, Cli, gwHosp)
:- hasAct(Pat, patient), aux2.6(), u0(Pat)
```

The set of active administrators $U_0 = \{\text{hpo1}, \text{pat1}\}$ is represented as facts $\text{u0}(\text{hpo1}), \text{u0}(\text{pat1})$ in the transformed policy. Similarly, for each *removeFact* permission rule $\text{permit}(U, \text{removeFact}(a)) :- \vec{L}$, the

```

memberOf(Cli, trCli(Pat, gwHosp))
:- consentTT(Pat, Cli, gwHosp), aux2.4()

permit(User, addFact(aux2.4()))
:- hasAct(User, pOfc(gwHosp))

memberOf(Cli, trCli(Pat, gwHosp))
:- hasAct(Cli, cli(gwHosp, Spcty)),
   memberOf(Cli, wkgp(W, gwHosp,
                    Spcty, WT)),
   encounter(EncID, Pat, W, gwHosp, Type),
   aux2.5()

permit(User, addFact(aux2.5()))
:- hasAct(User, pOfc(gwHosp))

permit(Pat, addFact(consentTT(Pat, Cli,
                             gwHosp)))
:- hasAct(Pat, patient), aux2.6() (4.1)

permit(User, addFact(aux2.6()))
:- hasAct(User, pOfc(gwHosp))

permit(Pat, removeFact(consentTT(Pat,
                                 Cli, gwHosp)))
:- hasAct(Pat, patient), aux2.7()

permit(User, addFact(aux2.7()))
:- hasAct(User, pOfc(gwHosp))

permit(Ag, addFact(consentTT(Pat, Cli,
                             gwHosp)))
:- hasAct(Ag, agent(Pat)), aux2.8()

permit(User, addFact(aux2.8()))
:- hasAct(User, pOfc(gwHosp))

permit(Ag, removeFact(consentTT(Pat, Cli,
                                 gwHosp)))
:- hasAct(Ag, agent(Pat)), aux2.9()

permit(User, addFact(aux2.9()))
:- hasAct(User, pOfc(gwHosp))

```

Fig. 3. Rules added to P_0 by the `simAddRule` transformation for the running example.

transformation adds the rule $!a \text{ :- } \vec{L}, u0(U)$. Let $\text{simAddRmFact}(P, U_0)$ denote the policy obtained by transforming policy P as described above, with set U_0 of active administrators.

The intention underlying the design of this transformation is that the set of atoms reachable in state graph $\text{SG}(\text{simAddRule}(P_0), U_0, T_{\text{-aR, -rR}})$ equals the set of atoms reachable in state graph $\text{SG}(\text{simAddRmFact}(\text{simAddRule}(P_0), U_0), U_0, T_{\text{-aR, -rR, -aF, -rF}})$, where $T_{\text{-aR, -rR, -aF, -rF}}$ is the transition relation without `addRule`, `removeRule`, `addFact`, or `removeFact` transitions. But then all transitions have been removed, so this is equivalent to the intention that the set of atoms reachable in the state graph $\text{SG}(\text{simAddRule}(P_0), U_0, T_{\text{-aR, -rR}})$ equals the set of atoms derivable in the policy $\text{simAddRmFact}(\text{simAddRule}(P_0), U_0)$. However,

the transformation does not quite achieve this goal—in other words, this equality does not quite hold—because the meaning of the original administrative permission rules differ from the meaning of the inference rules used to simulate them. For `addFact`, the original `addFact` permission rule means that a might (or might not) be added by an administrator when \vec{L} holds, while the added rule means that a necessarily holds (in the transformed policy) when \vec{L} holds. Similarly, for `removeFact`, the original `removeFact` permission rule means that a might (or might not) be removed by an administrator—causing $!a$ to hold—when \vec{L} holds, while the transformed rule means that $!a$ necessarily holds when \vec{L} holds. This change in the meaning of the rules affects the results of the tabling algorithm in phase 3, which is used to compute the atoms derivable from the transformed policy $\text{simAddRmFact}(\text{simAddRule}(P_0), U_0)$. Specifically, because phase 3 does not attempt to detect conflicts between negative subgoals and added facts or conflicts between positive subgoals and removed facts, it may produce derivations of atoms that are not actually derivable due to such conflicts (and are not reachable in $\text{SG}(\text{simAddRule}(P_0), U_0, T_{\text{-aR, -rR}})$). The overall algorithm is still sound, because phase 4 detects such conflicts in derivations of atoms and discards candidate solutions that involve those atoms.

Example: Figure 4 presents rules added to policy $\text{simAddRule}(P_0)$ by `simAddRmFact` transformation.

4.3 Phase 3: Tabled Policy Evaluation

Phase 3 is a modified version of Becker *et al.*'s algorithm for tabled evaluation with abduction [11] with the extension for proof graph construction [10]. We first present the original version of the algorithm and then describe our modifications.

4.3.1 Becker *et al.*'s Algorithm

Becker *et al.*'s algorithm appears in Figure 5. It uses resolution, extended to perform abduction. During resolution, when an attempt to prove a subgoal fails, if the subgoal is abducible, then it is assumed to be true, in which case it is said to be *abduced*, and the proof of the parent goal continues. The algorithm keeps track of abductions: each goal is associated with a set of abduced atoms on which it depends. The algorithm constructs a forest of proof trees, each consisting of a *root node*, intermediate *goal nodes*, and *answer nodes* as leaf nodes, defined as follows.

A *node* is either a *root node* $\langle G \rangle$, where G is an atom, or a tuple of the form $\langle G; \vec{Q}; S; \vec{c}; R; \Delta \rangle$, where G is an atom called the *index* (the goal whose derivation this node is part of), \vec{Q} is a list of subgoals that remain to be solved in the derivation of the goal, S is the partial answer (the instance of G that can be derived using the derivation that this node is part


```

memberOf(Cli, trCli(Pat, gwHosp))
:- consentTT(Pat, Cli, gwHosp), aux2.4()

aux2.4() :- u0(User), hasAct(User, pOfc(gwHosp))

memberOf(Cli, trCli(Pat, gwHosp))
:- hasAct(Cli, cli(gwHosp, Spcty)),
   memberOf(Cli, wkgp(W, gwHosp, Spcty, WT)),
   encounter(EncID, Pat, W, gwHosp, Type),
   aux2.5()

aux2.5() :- u0(User), hasAct(User, pOfc(gwHosp))

consentTT(hpo1, Cli, gwHosp)
:- u0(Pat), hasAct(Pat, patient), aux2.6()

aux2.6() :- u0(User), hasAct(User, pOfc(gwHosp))

!consentTT(hpo1, Cli, gwHosp)
:- u0(Pat), hasAct(Pat, patient), aux2.7()

aux2.7() :- u0(User), hasAct(User, pOfc(gwHosp))

consentTT(Pat, Cli, gwHosp)
:- u0(Ag), hasAct(Ag, agent(Pat)), aux2.8()

aux2.8() :- u0(User), hasAct(User, pOfc(gwHosp))

!consentTT(Pat, Cli, gwHosp)
:- u0(Ag), hasAct(Ag, agent(Pat)), aux2.9()

aux2.9() :- u0(User), hasAct(User, pOfc(gwHosp))

u0(hpo1)
u0(pat1)

```

Fig. 4. Rules added to the policy $\text{simAddRule}(P_0)$ by the simAddRmFact transformation for the running example.

of), \vec{c} is the list of child nodes of this node, R is the rule used to derive this node from its children in the derivation of S , and Δ is the residue (the set of atoms abduced in this derivation). Note that, in the definition of resolveClause in Figure 5, we use “abduction” as the name of the rule used to derive an abduced fact. If the list Q of subgoals is empty, the node is called an *answer node* with answer S . Otherwise, it is called a *goal node*, and the first atom in Q is its *current subgoal*. Each answer node is the root of a proof tree; goal nodes (representing queries) are not in proof trees. Selectors for components of nodes are: for $n = \langle G; \vec{Q}; S; \vec{c}; R; \Delta \rangle$, $\text{index}(n) = G$, $\text{subgoals}(n) = \vec{Q}$, $\text{pAns}(n) = S$, $\text{children}(n) = \vec{c}$, $\text{rule}(n) = R$, and $\text{residue}(n) = \Delta$.

Variable Ans contains the *answer table*, which is a partial function from atoms to sets of answer nodes. $\text{Ans}(G)$ contains all answer nodes for goal G found so far. Variable Wait contains the *wait table*, which is a partial function from atoms to sets of goal nodes. $\text{Wait}(G)$ contains all those nodes whose current subgoal is waiting for answers from $\langle G \rangle$. Whenever a new answer for $\langle G \rangle$ is produced, the computation

```

resolveClause( $\langle P \rangle$ )
1   $\text{Ans}(P) = \emptyset$ 
2  for ( $Q \leftarrow \vec{Q}$ ) in  $\text{Pol}$ 
3       $nd_1 = \langle P; Q :: \vec{Q}; Q; []; Q \leftarrow \vec{Q}; \emptyset \rangle$ 
4      if  $nd = \text{resolve}(nd_1, \langle P; []; P; []; \_; \emptyset \rangle)$  exists
5           $\text{processNode}(nd)$ 
6  if  $P$  is abducible
7       $\text{processAnswer}(\langle P; []; P; []; \text{abduction}; [P] \rangle)$ 

processAnswer( $nd$ )
1  match  $nd$  with  $\langle P; []; \_; \_; \_; \_ \rangle$  in
2      if there is no  $nd_0 \in \text{Ans}(P)$  such that  $nd \preceq nd_0$ 
3           $\text{Ans}(P) = \text{Ans}(P) \cup \{nd\}$ 
4          for  $nd'$  in  $\text{Wait}(P)$ 
5              if  $nd'' = \text{resolve}(nd', nd)$  exists
6                   $\text{processNode}(nd'')$ 

processNode( $nd$ )
1  match  $nd$  with  $\langle P; \vec{Q}; \_; \_; \_; \_ \rangle$  in
2      if  $\vec{Q} = []$ 
3           $\text{processAnswer}(nd)$ 
4      else match  $\vec{Q}$  with  $Q_0 :: \_$  in
5          if there exists  $Q'_0 \in \text{dom}(\text{Ans})$  such that
6               $Q_0$  is an instance of  $Q'_0$ 
7               $\text{Wait}(Q'_0) = \text{Wait}(Q'_0) \cup \{nd\}$ 
8              for  $nd'$  in  $\text{Ans}(Q'_0)$ 
9                  if  $nd'' = \text{resolve}(nd, nd')$  exists
10                   $\text{processNode}(nd'')$ 
11          else
12               $\text{Wait}(Q_0) = \{nd\}$ 
13               $\text{resolveClause}(\langle Q_0 \rangle)$ 

```

Auxiliary Definitions:

$$\langle G; []; S; \vec{c}; R; \Delta \rangle \preceq \langle G; []; S'; \vec{c}'; R'; \Delta' \rangle$$

$$\text{iff } |\Delta| \geq |\Delta'| \wedge (\exists \theta . S = S'\theta \wedge \Delta \supseteq \Delta'\theta)$$

for an answer node $n = \langle _; []; Q'; _; _; \Delta' \rangle$,
and Q'' and Δ'' fresh renamings of Q' and Δ' ,

$$\text{resolve}(\langle G; Q :: \vec{Q}; S; \vec{c}; R; \Delta \rangle, n) = \begin{cases} n' & \text{if unifiable}(Q, Q'') \\ & \text{where } \theta = \text{mostGeneralUnifier}(Q, Q'') \\ & n' = \langle G; \vec{Q}\theta; S\theta; [\vec{c}; n]; R; \Delta\theta \cup \Delta''\theta \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 5. Becker *et al.*'s algorithm for tabled policy evaluation with abduction and proof construction. $[]$ is the empty list. $x :: y$ is the list obtained by prepending an item x to list y . **match** exp **with** pat matches the value of expression exp with pattern pat and binds variables in pat accordingly.

involving these waiting nodes is resumed.

The auxiliary definitions in the lower part of Figure 5 define the subsumption relation \preceq on nodes and the resolve function. Intuitively, if $n \preceq n'$ (read “ n is subsumed by n' ”), then the answer node n provides

no more information about possible solutions than n' , so n can be discarded. $\text{resolve}(n, n')$ takes a goal node n and an answer node n' and combines the current subgoal of n with the answer provided by n' to produce a new node with fewer subgoals.

Constructors are not considered in [10], [11], but the algorithm can handle them, when the functions for matching and unification are extended appropriately.

The algorithm takes as input a query G , which is an atom, and the input policy Pol . The entry point is a call to $\text{resolveClause}(\langle G \rangle)$. The resolveClause function resolves clauses (i.e., rules) in the policy with the atom in a root node passed as argument. Starting from a root node $\langle P \rangle$, resolution with policy clauses produces goal nodes with index P . As the subgoals \bar{Q} are processed one by one, new P -indexed goal nodes are created with the remaining subgoals and with increasingly instantiated variants of P as partial answer. A proof branch ends when no subgoals remain, i.e., an answer node is generated.

4.3.2 Algorithm for Phase 3

This section describes our modified version of Becker *et al.*'s algorithm.

The algorithm considers three ways to satisfy a positive subgoal: through a fact or rule in the policy, through addition of a fact using a transformed addFact permission rule (this does not require a separate case in the algorithm, because these rules are handled in the same way as other rules), and through abduction (i.e., by assumption that the subgoal holds in the initial policy and still holds when the rule containing it as a premise is evaluated). The algorithm considers two ways to satisfy a negative subgoal: through removal of a fact using a transformed removeFact permission rule (again, this does not require a separate case in the algorithm) and through abduction (i.e., by assumption that the negative subgoal holds in the initial policy and still holds when the rule containing it as a premise is evaluated).

The algorithm can abduce a negative extensional literal $!a$ when this is consistent with the initial policy, in other words, when a is not in P_0 . To enable this, in the definition of resolveClause , we replace " P is abducible" with " $P \in \llbracket A \rrbracket \vee (\exists a \in \text{Atom}_{ex}. a \notin P_0 \wedge P \text{ is } !a)$ ", where Atom_{ex} is the set of extensional atoms. If a is not ground, disequalities in d_{init} in phase 4 will ensure that the solution includes only instances of a that are not in P_0 .

The tabling algorithm treats the negation symbol "!" as part of the predicate name; in other words, it treats p and $!p$ as unrelated predicates. Phase 4 interprets "!" as negation and checks appropriate consistency conditions relating positive and negative facts.

Wildcards do not need special treatment in this phase. To establish through abduction a negative premise that contains wildcards, the negative literal is simply abducted (with wildcards in it) into the

residue. Recall from Section 2.1 that wildcards can be used in a negative literal $!p(\dots)$ only if there are no removeFact permission rules for p . This implies we do not need to consider how to establish negative literals containing wildcards using removals of facts.

The definition of resolve in Figure 5 checks whether $\text{unifiable}(Q, Q')$ holds and, if so, computes the residue of the resolve node n' to be $\Delta\theta \cup \Delta''\theta$. Since we, unlike Becker *et al.*, allow specification of a set nAb of not-abducible terms (which might overlap with the set Ab), instantiating a term in the residue can move it from $\llbracket Ab \rrbracket$ to $\llbracket nAb \rrbracket$, causing it not to be abducible. Therefore, in the definition of resolve , we replace the condition $\text{unifiable}(Q, Q')$ with the condition $\text{unifiable}(Q, Q') \wedge (\Delta\theta \cup \Delta''\theta \subseteq \llbracket A \rrbracket)$. It suffices to consider only instantiation with the most general unifier, because nAb is closed under instantiation.

Becker *et al.*'s algorithm explores all derivations for a goal except that the subsumption check in processAnswer in Figure 5 prevents use of the derivation represented by answer node n from being added to the answer table and thereby used as a sub-derivation of a larger derivation if the partial answer in n is subsumed by the partial answer in an answer node n' that is already in the answer table. However, the larger derivation that uses n' as a derivation of a subgoal might turn out to be infeasible (i.e., have unsatisfiable ordering constraints) in phase 4, while the larger derivation that uses n as a derivation of that subgoal might turn out to be feasible. We adopt the simplest approach to overcome this problem: we replace the subsumption relation \preceq in processAnswer method with the α -equality relation $=_\alpha$, causing the tabling algorithm to explore all derivations of goals. α -equality is equality modulo renaming of variables that do not appear in the query's top-level goal G_0 .

An undesired side-effect of this change is that the algorithm may get stuck in a cycle in which it repeatedly uses some derivation of a goal as a sub-derivation of a larger derivation of the same goal. Exploring such derivations is unnecessary, because the algorithm is not required to find a representation of all sequences of administrative actions that reach the goal. Specifically, if the algorithm has already constructed a node n , then it is unnecessary for the algorithm to construct a node n' that has the same index, partial answer, and residue as n and a proof graph that contains n . Therefore, we modify the definition of resolve as follows, so that the algorithm does not generate a node n' corresponding to the latter derivation: we replace $\text{unifiable}(Q, Q')$ with $\text{unifiable}(Q, Q') \wedge \text{noCyclicDeriv}(n')$, where

$$\begin{aligned} \text{noCyclicDeriv}(n') = & \exists d \in \text{proof}(n'). \text{isAns}(d) \\ & \wedge \langle \text{index}(d), \text{pAns}(d), \text{residue}(d) \rangle =_\alpha \\ & \langle \text{index}(n'), \text{pAns}(n'), \text{residue}(n') \rangle \end{aligned}$$

where the *proof* of a node n , denoted $\text{proof}(n)$, is the set of nodes in the proof graph rooted at node

n , i.e., $\text{proof}(n) = \{n\} \cup \bigcup_{n' \in \text{children}(n)} \text{proof}(n')$, and where $\text{isAns}(n)$ holds iff n is an answer node. $\text{noCyclicDeriv}(n')$ does not check whether $\text{rule}(n') = \text{rule}(d)$ or $\text{subgoals}(n') = \text{subgoals}(d)$, because exploration of n' is unnecessary, by the above argument, regardless of the values of $\text{rule}(n')$ and $\text{subgoals}(n')$.

The *partial answer substitution* for a node n , denoted $\theta_{\text{pa}}(n)$, is the substitution that, when applied to $\text{index}(n)$, produces $\text{pAns}(n)$. We extend the algorithm to store $\theta_{\text{pa}}(n)$ in each node n , as follows. In the `resolveClause` method, the θ_{pa} component in both nodes passed to `resolve` is the empty substitution. In the `resolve` function, $\theta_{\text{pa}}(n')$ is $\theta \circ \theta_{\text{fr}} \circ \theta_{\text{pa}}(n_1)$, where θ_{fr} is the substitution that performs the fresh renaming of Q' and Δ' , n_1 denotes the first argument to `resolve`, and \circ denotes composition of substitutions.

In summary, for an abductive atom-reachability query of the form in Section 3, phase 3 applies the algorithm for tabled policy evaluation with abduction and proof construction, modified as described above, to the policy $\text{simAddRmFact}(\text{simAddRule}(P_0), U_0)$ with the given goal G_0 and specification A of abducible atoms.

Example: Figures 6 and 7 in Section 10 in the supplemental material present proofs ψ_1 and ψ_2 generated for the example query in Section 3.1.

4.4 Phase 4: Ordering Constraints

Phase 4 considers constraints on the execution order of administrative operations. An *administrative node* is a node n such that $\text{rule}(n)$ is a transformed `addFact` or `removeFact` permission rule. The ordering must ensure that, for each administrative node or goal node n , (1) each administrative operation n' used to derive n occurs before n (this is a *dependence constraint*) and its effect is not undone by a conflicting operation that occurs between n' and n (this is an *interference-freedom constraint*), and (2) each assumption about the initial policy on which n relies is not undone by an operation that occurs before n (this is also an *interference-freedom constraint*). When generating the ordering constraints in item (1) for node n , administrative operations used to derive n' are not considered, because the derivation of n does not (directly) depend on the effects of those operations; n depends on those operations only via the fact that they permit n' , and ordering constraints that ensure they permit n' are generated when item (1) is considered for node n' . The concept of interference freedom originated in work on Hoare logics for concurrent programs, and dependence constraints are analogous to condition synchronization [13].

A straightforward but inefficient algorithm would enumerate each permutation of the set of administrative operations (corresponding to the administrative nodes) in each proof graph from phase 3 and check whether it satisfies the ordering constraints. We adopt a more efficient approach in which the ordering con-

straints for each proof graph are represented symbolically and tested for satisfiability. The overall ordering constraint is represented as a Boolean combination of labeled ordering edges. A labeled ordering edge is a tuple $\langle n, n', D \rangle$, where the label D is a conjunction of tuple disequalities or false, with the interpretation: n must precede n' , unless D holds. If D holds, then n and n' operate on distinct atoms, so they commute, so the relative order of n and n' is unimportant.

Phase 4 iterates over the answer nodes from Phase 3. For each answer node, it generates a conjunction of dependence constraints and interference-freedom constraints, puts the resulting Boolean expression in DNF, and then checks, for each disjunct c , whether the ordering constraints in c can be satisfied, i.e., whether they are acyclic. If so, the disequalities labeling the ordering constraints do not need to be included in the solution. However, if the generated ordering constraints are cyclic, then the algorithm removes a minimal set of ordering constraints to make the remaining ordering constraints acyclic (by computing the set of all cycles in the ordering constraints in c and removing one edge from each cycle), and includes the disequalities that label the removed ordering constraints in the solution. After satisfiability of the constraints has been checked (including the consistency constraint that each abduced negative literal holds initially and still holds when the rule containing it as a premise is evaluated), negative literals are removed from the residue; this is acceptable, because the problem definition asks for a representation of only minimal-residue ground solutions, not all ground solutions (negative literals provide information about which sets of positive literals that are supersets of the set of positive literals in the residue are also solutions to the query. Pseudocode for Phase 4 and example ordering constraints are in Section 11 in the supplemental material.

Repeated Administrative Operations: The tabling algorithm in phase 3 re-uses nodes, including administrative nodes. This makes the analysis more efficient and avoids unnecessary repetition of administrative operations in plans. However, in some cases, administrative operations need to be repeated; for example, it might be necessary to add a fact, remove it, and then add it again, in order to reach a goal. The version of our algorithm described above cannot generate plans with repeated administrative operations, but it does identify when repeated operations might be necessary, using the function `mightNeedRepeatedOp`, and returns a message indicating this. `mightNeedRepeatedOp(n_g, c)` returns true iff some node n in c is a child of multiple nodes in $\text{proof}(n_g)$; in such cases, it might be necessary to “split” n —i.e., replace n with multiple nodes, one for each parent—in order to satisfy ordering constraints. We sketch here how to extend the algorithm to generate plans with

repeated administrative operations (the extension is not needed for the running example or the case study in Section 5). A new variable *Split* stores the set of nodes that need to be split. A node n in *Split* is identified by the contents of the nodes on the path from the node to the root node n_g (including the contents of n and n_g). *Split* is initialized to \emptyset . If `mightNeedRepeatedOp(n_g, c)` returns true, for each node n in c that is a child of multiple nodes in `proof(n_g)`, each path from n to n_g is added to *Split*, and phase 3 is re-run. The tabling algorithm in Phase 3 is modified so that nodes in *Split* are not re-used; specifically, in function `processAnswer(nd)` in Figure 5, if the path from nd to the root is in *Split*, then nd is not added to the answer table in line 3. After phase 3 is re-run, the algorithm continues as usual to phase 4. Phases 3 and 4 might be iterated multiple times, until all nodes that caused `mightNeedRepeatedOp` to return true have been split into multiple nodes.

4.5 Termination and Running Time

We consider the termination and running time of each phase. Phases 1 and 2 are fast linear-time transformations of the input. Phase 3 can diverge. The possibility of non-termination is inherited from Becker *et al.*'s algorithm. It is intrinsic to the problem, in the sense that there are abductive queries involving recursive rules such that every comprehensive solution is an infinite set [10]. In the context of access control, such recursive rules might arise in policies that allow unbounded delegation chains. Becker *et al.* give a static condition—absence of recursive rules with a certain structure—that ensures termination. For policies not satisfying this condition, they give some pragmatic strategies for ensuring termination, e.g., modifying the algorithm to return only solutions containing at most a specified number of abduced atoms.

Phase 4 has two potentially expensive steps: putting the ordering constraint in DNF, and computing the set of all cycles in the ordering constraints in a disjunct. Putting a formula in DNF takes exponential time in the worst case. In practice, the formulas involved are typically not large, because typically most pairs of nodes in a proof graph do not conflict. Computing the set of all cycles in a graph takes at least factorial time in the worst case, because a complete graph with n nodes contains more than $(n - 1)!$ cycles. The algorithm we use for computing the set of cycles in a graph [14] takes $O((|V| + |E|)(c + 1))$ time, where V and E are the sets of nodes and edges in the graph, respectively, and c is the number of cycles in the graph. In practice, the number of cycles in the ordering constraints in a disjunct is typically small, so computing the set of cycles is not expensive.

4.6 Correctness

The algorithm is correct in the sense that, when it terminates with a solution, it returns a comprehen-

sive solution to the given abductive atom-reachability query. A proof of correctness appears in Section 9 in the supplemental material. The algorithm is incomplete, because it might diverge, as discussed above. Incompleteness is expected, because the abductive atom-reachability problem is undecidable. Also, without the extension to handle repeated administrative operations, the algorithm might indicate that repeated administrative operations might be needed, instead of returning a solution, as discussed in Section 4.4.

5 EXPERIENCE

To gain experience with the framework and analysis, we wrote a policy for a healthcare network in ACAR, implemented the analysis algorithm in OCaml, and used the implementation to evaluate a few queries. A detailed presentation of the healthcare network policy appears in [15, Chapter 3]. The policy controls permissions for registration of users (patients, clinicians, etc.), workgroup management (creating workgroups, and adding and removing members), agent management (patients appointing agents), consent to treatment (patients or their agents granting consent to treatment by a specified clinician), encounter management (creating an encounter, in which a patient is treated by a workgroup, and closing an encounter), and access to patient health records. The policy consists of 22 rules. Healthcare networks are interesting from the perspective of policy administration, because they involve policies at several organizational levels. Our case study involves policies at the levels of the healthcare network itself, a prototypical hospital (`gWHosp`) in the network, a prototypical substance abuse facility (`gCSAF`) in the network, and workgroups in those facilities. For example, users at the facility level can modify the facility's policy in ways consistent with the administrative permissions provided by the healthcare network's policy. An example of how a facility policy officer might specialize a facility's policy for appointing clinicians appears in Section 2.4. As another example, rules added by the substance abuse facility's policy officer impose stricter conditions for access to patient health records than corresponding rules added by the hospital's policy officer; specifically, the former rules allow a clinician to access a patient's health records only if the clinician has been individually granted consent to treatment by the patient, while the latter rules allow a clinician access to a patient's health records if the patient has granted consent to treatment by a workgroup of which the clinician is a member.

One sample query has initial policy $P_0 = P_{\text{HCN}} \cup P_{\text{U1}}$, where P_{HCN} is the healthcare network policy and P_{U1} contains a few facts about the prototypical users `hpo1`, a member of `pOfc(gWHosp)`, `clin1`, a clinician at `gWHosp`, and `user1`, a user with no roles (the only

fact about `user1` in P_{U_1} is `user(user1)`. The other components of the query are $U_0 = \{\text{hpo1}, \text{user1}\}$, $Ab = \{\text{memberOf}(\text{User}, \text{wkgp}(\text{WG}, \text{gwHosp}, \text{Spcty}, \text{team}))\}$, $nAb = \{\}$, $A = \langle Ab, nAb \rangle$, and $G_0 = \text{workgroupHead}(\text{GoalUser}, \text{cardioTeam}, \text{gwHosp})$. The analysis generates 1493 nodes and returns five solutions. For example, one solution has partial answer `workgroupHead(GoalUser, cardioTeam, gwHosp)`, residue `\{memberOf(GoalUser, wkgp(cardioTeam, gwHosp, Spcty, team))\}`, and tuple disequality `\langle GoalUser \rangle \neq \langle hpo1 \rangle`. The disequality reflects that `hpo1` can appoint himself to the `hrManager(gwHosp)` role, can then appoint himself and other users as members of `cardioTeam`, and can then appoint other users as team head, but cannot then appoint himself as team head, because the rule that allows HR managers to appoint workgroup heads contains a negative premise that prohibits an HR manager from appointing a head for a workgroup if that HR manager is a member of that workgroup.

Another sample query has initial policy $P_0 = P_{\text{HCN}} \cup P_{U_2}$, where P_{U_2} contains a few facts about the prototypical users `hpo1`, a member of `pOfc(gwHosp)`, `hhr`, a member of `hrManager(gwHosp)`, `clil`, a clinician at `gwHosp`, and `pat1`, a patient at `gwHosp`. The other components of the query are $U_0 = \{\text{hpo1}, \text{hhr}\}$, $Ab = \{\text{memberOf}(\text{User}, \text{wkgp}(\text{WG}, \text{gwHosp}, \text{Spcty}, \text{team})), \text{encounter}(\text{EncID}, \text{pat1}, \text{Wkgp}, \text{gwHosp}, \text{Type})\}$, $nAb = \{\}$, $A = \langle Ab, nAb \rangle$, and $G_0 = \text{memberOf}(\text{clil}, \text{trCli}(\text{pat1}, \text{gwHosp}))$. Informally, the query asks whether a clinician can become the treating clinician for a patient at `gwHosp` through actions of the policy officer and HR manager (without actions of the patient or the clinician). The analysis generates 4946 nodes and returns one solution with residue `\{memberOf(clil, wkgp(WG, gwHosp, Spcty, team), encounter(EncID, pat1, WG, gwHosp, Type))\}`, which indicates that this is possible if `hhr` makes `clil` a member of a workgroup `WG` that is currently handling an encounter `EncID` for `pat1`. This illustrates that the analysis can bring non-obvious and possibly unanticipated scenarios to the attention of policy auditors.

Running time: We ran the algorithm on a `hp dc7900` with 3.0 GHz Intel Core2 Duo processor. The above examples take 0.20 seconds and 2.40 seconds, respectively, of user+system time. We used the GNU profiler, `gprof`, with a sampling period of 0.01 seconds, to help measure the cost of each phase. Phases 1 and 2 are fast linear-time transformations of the input, so phases 3 and 4 dominate the running time. For the first example, phase 4 consumes 38% of the running time, and phase 3 consumes most of the remainder. For the second example, phase 4 consumes less than

1% of the running time, and phase 3 consumes most of the running time. In both examples, the steps in phase 4 with high worst-case asymptotic time complexity—putting a formula in DNF and finding all cycles in a graph—take negligible time (`gprof` reports it as 0).

Multiple factors contribute to the algorithm’s speed. Policy rules are relatively small compared to databases of policy-related facts, so the input to our abductive analysis algorithm is relatively small. Our algorithm is goal-directed, so it avoids exploring irrelevant possibilities. Our algorithm avoids exploring permutations of the administrative operations in a proof graph by constructing and checking satisfiability of ordering constraints.

6 RELATED WORK

6.1 Administration of Rule-Based Policies

Our work is inspired by Becker *et al.*’s abductive policy analysis for rule-based policy languages [10], [11]. The main difference between their work and ours is that they do not consider changes to the rules. Also, they do not consider constructors and negation, while ACAR allows constructors and allows negation applied to extensional predicates. Becker and Nanz’s earlier work [16], [9] also considers changes to the facts with fixed rules but does not consider abductive analysis: it assumes the initial policy is known.

DynPAL is an administrative framework with a rule-based access control policy language [7]. DynPAL allows stratified negation of intensional predicates; ACAR does not. DynPAL provides more complex administrative operations than ACAR for adding and removing facts. On the other hand, DynPAL’s administrative framework considers only addition and removal of facts, not addition or removal of rules. Becker *et al.* also present two methods for reachability analysis for DynPAL [7]. The first method, based on AI planning, it requires the domain of constants in the language to be finite and the policy to be *tight*, i.e., every rule defining an intensional predicate can be finitely unfolded down to extensional predicates. The second method, based on theorem proving, does not require finite domains but requires tightness. Both methods deal only with addition and removal of facts and solve reachability from a given initial state. Our abductive analysis does not require the domain of constants to be finite, or the policy to be tight, and returns solutions that are more general in terms of the facts in the initial policy.

Craven *et al.* present a rule-based policy language with an administrative framework based on Event Calculus [8]. Their policy language is richer than ACAR in that it supports obligations, time constraints, and stratified negation. They describe how to use abductive logic programming to solve a variety of policy analysis problems. They do not consider addition and removal of rules, and their analysis algorithm

restricts abduction to ground residues. In contrast, we consider addition and removal of rules and facts, and our analysis algorithm supports non-ground residues.

Barletta *et al.* [17] give a model checking algorithm for bounded-length reachability for Access Control Systems (ACSs). In their work, the set of rules is fixed, and abductive analysis is not considered.

6.2 Administration of RBAC

ARBAC97 is the first comprehensive administrative framework for RBAC [2]. ACAR is more expressive than ARBAC97 in many ways (rule (2.5) in Figure 2 is a good example of a policy that cannot be expressed in ARBAC97) but is also less expressive in some ways, as discussed in Section 12 in the supplemental material.

Analysis algorithms for user-permission reachability for ARBAC97 and variants thereof have been developed, e.g., [3], [4], [5], [6]. Analysis of ARBAC considers, in effect, only addition and removal of facts, not rules, because administrative operations in ARBAC (e.g., removing a user from a role) correspond to addition and removal of facts. Work on analysis of ARBAC generally does not consider abductive analysis, with the exception of some works, such as [4], that consider the weakest precondition problem for ARBAC97, which asks for minimal sets of initial role memberships of the target user that allow the target user to be eventually assigned to given roles.

7 CONCLUSIONS AND FUTURE WORK

This paper’s main contribution is the first analysis algorithm for a rule-based policy framework with administrative policies that control changes to the rules as well as the facts in the policy. Furthermore, through the use of abduction, the analysis algorithm can analyze policies even when only partial information about the facts in the initial policy is available. Directions for future work include relaxing the restrictions on use of wildcard and negation, and developing an analysis algorithm based on state-space exploration, instead of tabling, for the comprehensive abductive atom-reachability problem. The main challenge for the latter is how to handle the lack of complete information about the initial policy.

ACKNOWLEDGMENTS

This work was supported in part by ONR Grant N00014-07-1-0928, NSF Grants CNS-0831298 and CCF-1248184, and AFOSR Grant FA0550-09-1-0481. We thank the reviewers for helpful comments.

REFERENCES

[1] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, “Role-based access control models,” *IEEE Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[2] R. Sandhu, V. Bhamidipati, and Q. Munawer, “The ARBAC97 model for role-based administration of roles,” *ACM Transactions on Information and Systems Security*, vol. 2, no. 1, pp. 105–135, Feb. 1999.

[3] N. Li and M. V. Tripunitara, “Security analysis in role-based access control,” *ACM Transactions on Information and System Security*, vol. 9, no. 4, pp. 391–420, Nov. 2006.

[4] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman, “Efficient policy analysis for administrative role based access control,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 2007.

[5] A. Sasturkar, P. Yang, S. D. Stoller, and C. R. Ramakrishnan, “Policy analysis for administrative role based access control,” *Theoretical Computer Science*, vol. 412, no. 44, pp. 6208–6234, Oct. 2011.

[6] S. D. Stoller, P. Yang, M. Gofman, and C. R. Ramakrishnan, “Symbolic reachability analysis for parameterized administrative role based access control,” *Computers & Security*, vol. 30, no. 2-3, pp. 148–164, March-May 2011.

[7] M. Y. Becker, “Specification and analysis of dynamic authorisation policies,” in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 2009, pp. 203–217.

[8] R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, and A. Bandara, “Expressive policy analysis with enhanced system dynamicity,” in *Proc. 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*. ACM, 2009, pp. 239–250.

[9] M. Y. Becker and S. Nanz, “A logic for state-modifying authorization policies,” *ACM Transactions on Information and System Security*, vol. 13, no. 3, 2010.

[10] —, “The role of abduction in declarative authorization policies,” in *Proc. 10th International Conference on Practical Aspects of Declarative Languages (PADL 2008)*, ser. Lecture Notes in Computer Science, vol. 4902. Springer, 2008, pp. 84–99.

[11] M. Y. Becker, J. F. Mackay, and B. Dillaway, “Abductive authorization credential gathering,” in *Proc. IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*. IEEE Computer Society, Jul. 2009, pp. 1–8.

[12] P. Gupta, S. D. Stoller, and Z. Xu, “Abductive analysis of administrative policies in rule-based access control,” in *Proc. Seventh International Conference on Information Systems Security (ICISS 2011)*. Springer, Dec. 2011, pp. 116–130.

[13] F. B. Schneider, *On Concurrent Programming*. Springer, 1997.

[14] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[15] P. Gupta, “Abductive analysis of administrative policies in rule-based access control,” Ph.D. dissertation, Stony Brook University, Dec. 2011.

[16] M. Y. Becker and S. Nanz, “A logic for state-modifying authorization policies,” in *Proc. 12th European Symposium On Research In Computer Security (ESORICS)*, 2007, pp. 203–218.

[17] M. Barletta, S. Ranise, and L. Viganò, “Automated analysis of scenario-based specifications of distributed access control policies with non-mechanizable activities,” in *Proc. 8th International Workshop on Security and Trust Management (STM)*. Springer, 2012, pp. 49–64.

Puneet Gupta Puneet Gupta received the B.Tech. degree in Computer Science from IIT Delhi in 2006 and the Ph.D. degree in Computer Science from Stony Brook University in 2011. He is currently a Software Engineer at Google, Inc.

Scott D. Stoller Scott D. Stoller received the B.A. degree in Physics, summa cum laude, from Princeton University in 1990 and the Ph.D. degree in Computer Science from Cornell University in 1997. He is currently a Professor at Stony Brook University.

Zhongyuan Xu Zhongyuan Xu received the B.S. and M.S. degrees in Computer Science from Peking University in 2006 and 2009, respectively. He is currently a Ph.D. candidate at Stony Brook University.

8 PROOF OF UNDECIDABILITY OF ABDUCTIVE ATOM-REACHABILITY FOR ACAR

This section briefly describes a reduction from the user-role reachability problem for PARBAC without role hierarchy, which is known to be undecidable [6], to the abductive atom-reachability problem for ACAR.

PARBAC, short for “Parameterized ARBAC”, is based on ARBAC97, extended with parameters. A PARBAC policy is defined by a set RS of role schemas, a set U of users, a can_assign relation, and a can_revoke relation. A role schema consists of a role name and a list of named parameters. The can_assign relation contains tuples of the form $\langle r_a, \langle P, N \rangle, r \rangle$, where r_a is an administrative role, r is a regular (i.e., non-administrative) role, and P and N are sets of regular roles called the *positive precondition*, and *negative precondition*, respectively. The meaning of such a tuple is that a member of r_a can assign a user to be a member of r , if that user is a member of all of the roles in P and none of the roles in N . The can_revoke relation contains tuples of the form $\langle r_a, r \rangle$, where r_a is an administrative role, and r is a regular role. The meaning of such a tuple is that a member of r_a can remove users from r . Since there is no role hierarchy, we omit the `directMemberOf` predicate and use `memberOf` as the extensional predicate recording role membership. A tuple $\langle r_a, \langle \{p_1, \dots, p_i\}, \{n_1, \dots, n_j\} \rangle, r \rangle$ in can_assign is translated into the ACAR rule

```
permit (Uadmin, addFact (memberOf (U, r)))
:- memberOf (Uadmin, r_a), memberOf (U, p_1),
   ..., memberOf (U, p_i),
   !memberOf (U, n_1), ..., !memberOf (U, n_j)
```

A tuple $\langle r_a, r \rangle$ in can_revoke is translated into the ACAR rule

```
permit (Uadmin, addFact (memberOf (U, r)))
:- memberOf (Uadmin, r_a).
```

For example, the can_assign tuple $\langle Chair(dept = D), \langle \{Faculty(dept = D)\}, \{UgradAdComm(dept = D)\} \rangle, GradAdComm(dept = D) \rangle$ specifies that the Chair of department D can assign faculty of that department to the department’s graduate admissions committee if they are not members of the department’s undergraduate admissions committee. For readability, PARBAC uses named parameters, but this does not affect the expressiveness, so we translate them to positional parameters in ACAR. The above tuple is translated to the ACAR rule

```
permit (Uadmin, addFact (memberOf (U,
                                   GradAdComm(D))))
:- memberOf (Uadmin, Chair(D)),
   memberOf (U, Faculty(D)),
   !memberOf (U, UgradAdComm(D))
```

A user-role reachability query for PARBAC asks whether a given user u can become a member

of a given set $\{r_1, \dots, r_i\}$ of roles, given the initial role memberships of the user and a PARBAC policy. This is translated into an abductive atom-reachability query with an ACAR policy containing the translations of the can_assign and can_revoke tuples, `memberOf` facts expressing the user’s initial role memberships, and a rule `goal() :- memberOf(u, r_1), ..., memberOf(u, r_n)`. The atom `goal()` is the goal of the query, and no atoms are abducible.

9 PROOF OF CORRECTNESS OF THE ANALYSIS ALGORITHM

Correctness of Phase 1: We prove correctness of phase 1 in two steps. We define a transition relation $T_{\downarrow aR, -rR}$ that contains a restricted set of `addRule` transitions and no `removeRule` transitions. The first step of the proof, embodied in Theorem 9.3, shows that an atom is reachable in $SG(P_0, U_0, T)$ iff it is reachable in $SG(P_0, U_0, T_{\downarrow aR, -rR})$. The second step of the proof, embodied in Theorem 9.6, shows that an atom is reachable in $SG(P_0, U_0, T_{\downarrow aR, -rR})$ iff it is reachable in $SG(\text{simAddRule}(P_0), U_0, T_{-aR, -rR})$.

The *restricted-addRule, no-removeRule transition relation* $T_{\downarrow aR, -rR}$ is defined the same way as the transition relation T in Section 2.4 except (1) `addRule` transitions are restricted so that they do not add additional premises to rule patterns, and (2) `removeRule` transitions are eliminated.

Policy P is *at least as strict as* policy P' , denoted $P \leq P'$, if (1) P and P' contain the same set of (explicitly given) facts, and (2) for every rule R in P , P' contains a rule R' such that R is at least as strict as R' . Recall that the *at-least-as-strict-as* relation on rules is defined in Section 2.4.

Lemma 9.1. For all policies P and P' , if $P \leq P'$, then $\llbracket P \rrbracket \subseteq \llbracket P' \rrbracket$.

Proof: The proof is by induction on the derivation of the fact, considered as a tree built from rules and facts. The proof relies on the restriction that negation is applied only to extensional predicates. Consider a derivation of a fact f from P . f can be derived from P' using the same derivation except with each rule R in P replaced with a rule R' in P' such that R is at least as strict as R' . We need to show that each premise q of R' , instantiated using the same substitution used to instantiate R in the derivation of f , holds in P' . Since R is at least as strict as R' , the instance of R used in the derivation of f has the same premise q , and q holds in P . If q is a positive premise, then the derivation of q from P is a sub-derivation of the derivation of f , so by the induction hypothesis, q is derivable from P' . If q is a negative premise $!a$, then a must be an atom for an extensional predicate, so it suffices to consider the facts that appear explicitly in P and P' (i.e., we do not need to consider the possibility of deriving a using rules). Since q holds in P , a does not appear in

P . Since $P \leq P'$ implies that P and P' contain the same set of facts, a does not appear in P' , so q holds in P' . \square

Lemma 9.2. For every policy P_0 and set U_0 of users, for every policy P in $\text{SG}(P_0, U_0, T)$, there exists a policy P' in $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ such that $P \leq P'$.

Proof: Let $p_i = P_0 \xrightarrow{u_0:op_0} P_1 \xrightarrow{u_1:op_1} \dots \xrightarrow{u_{n-1}:op_{n-1}} P_n$ be a path in $\text{SG}(U_0, P_0)$ from the initial policy to P , hence $P_n = P$. We show by construction that there is a corresponding path $p'_i = P_0 \xrightarrow{u'_0:op'_0} P'_1 \xrightarrow{u'_1:op'_1} \dots \xrightarrow{u'_{n-1}:op'_{n-1}} P'_n$ in $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ such that for each $i \in [0 \dots n]$, $P_i \leq P'_i$.

To simplify the correspondence, we allow skip transitions in p'_i .

Based on the definition of the transition relation, if op_i is an `addRule` transition `addRule(R_i)`, then there exists an `addRule` permission rule R_i^{arp} in P_i (“arp” is mnemonic for “addRule permission”) and a rule R'_i such that P_i derives `permit(u_i , addRule(R'_i))` using R_i^{arp} in the last step of the derivation, and R_i is stricter than R'_i (note that R'_i already reflects instantiations of variables).

p'_i is defined as follows.

$$u'_i = u_i$$

$$op'_i = \begin{cases} op_i & \text{if } op_i \text{ has the form} \\ & \text{addFact}(\dots) \text{ or} \\ & \text{removeFact}(\dots) \\ \text{skip} & \text{if } op_i \text{ has the form} \\ & \text{removeRule}(\dots) \\ \text{addRule}(R'_i) & \text{if } op_i \text{ has the form} \\ & \text{addRule}(R_i), \\ & \text{where } R'_i \text{ is defined above} \end{cases}$$

We prove by induction on i that (a) $P_i \leq P'_i$ and (b) `permit(u'_i , op'_i)` \in $\llbracket P'_i \rrbracket$.

Base case: In the base case, $i = 0$.

(a) $P_0 \leq P_0$ follows directly from the definition of \leq .

(b) We need to show `permit(u'_0 , op'_0)` \in $\llbracket P_0 \rrbracket$.

case: op_0 is `removeRule`. This case is trivial, because op'_0 is `skip`, and (as a special case) `skip` is always permitted.

case: op_0 is `addFact` or `removeFact`. op'_0 is the same as op_0 , so `permit(u'_0 , op'_0)` \in $\llbracket P_0 \rrbracket$ follows directly from `permit(u_0 , op_0)` \in $\llbracket P_0 \rrbracket$.

case: op_0 is `addRule(R_0)`. In this case, op'_0 is `addRule(R'_0)`. The definition of R'_i directly implies that `permit(u_0 , addRule(R'_0))` \in $\llbracket P_0 \rrbracket$.

Step case: In the step case, we assume $P_i \leq P'_i$ and `permit(u'_i , op'_i)` \in $\llbracket P'_i \rrbracket$.

(a) We need to show $P_{i+1} \leq P'_{i+1}$. This follows directly from the induction hypothesis and the definitions of op'_i and \leq . Note that this proof does

not rely on the claim that (b) holds in the step case, so this conclusion can be used in the following proof of (b).

(b) We need to show `permit(u'_{i+1} , op'_{i+1})` \in $\llbracket P'_{i+1} \rrbracket$.

case: op_{i+1} is `removeRule`. This case is trivial, because op'_{i+1} is `skip`, and (as a special case) `skip` is always permitted.

case: op_{i+1} is `addFact` or `removeFact`. op'_{i+1} is the same as op_{i+1} , so `permit(u'_{i+1} , op'_{i+1})` \in $\llbracket P'_{i+1} \rrbracket$ follows directly from `permit(u_{i+1} , op_{i+1})` \in $\llbracket P_{i+1} \rrbracket$, $P_{i+1} \leq P'_{i+1}$, and Lemma 9.1.

case: op_{i+1} is `addRule(R_{i+1})`. In this case, op'_{i+1} is `addRule(R'_{i+1})`, as defined above. The definition of R'_i in that paragraph directly implies that `permit(u_{i+1} , addRule(R'_{i+1}))` \in $\llbracket P_{i+1} \rrbracket$. This, together with $P_{i+1} \leq P'_{i+1}$ and Lemma 9.1, imply that `permit(u_{i+1} , addRule(R'_{i+1}))` \in $\llbracket P'_{i+1} \rrbracket$. \square

Theorem 9.3. For every policy P_0 , set U_0 of users, and atom a , $\text{SG}(P_0, U_0, T)$ contains a policy P with $a \in \llbracket P \rrbracket$ iff $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ contains a policy P' with $a \in \llbracket P' \rrbracket$.

Proof: We prove one direction at time.

Suppose $\text{SG}(P_0, U_0, T)$ contains a policy P such that $a \in \llbracket P \rrbracket$. We need to show that there exists P' in $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ such that $a \in \llbracket P' \rrbracket$. This follows immediately from Theorem 9.2, the definition of \leq , Lemma 9.1, and the fact that goals do not contain negative intensional literals.

Suppose $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ contains a policy P' such that $a \in \llbracket P' \rrbracket$. We need to show that $\text{SG}(P_0, U_0, T)$ contains a policy P such that $a \in \llbracket P \rrbracket$. This follows from the fact that the restricted transition relation $T_{\downarrow aR, -rR}$ is a subset of the transition relation T , which implies that $\text{SG}(P_0, U_0, T)$ also contains P . \square

For the second step of the proof of correctness of phase 1, we start by showing that $\text{SG}(P_0, U_0, T_{\downarrow aR, -rR})$ is similar to $\text{SG}(\text{simAddRule}(P_0), U_0, T_{-aR, -rR})$ in the sense captured in Theorem 9.6. We call predicates with names like `aux $_R$` as *auxiliary predicates*. We assume that the original policy does not contain predicates with such names.

We give a more formal definition of the similarity relationship between policies (an informal definition appears in Section 4.1). Formally, $P \simeq P'$ if P contains no occurrences of auxiliary predicates, and there exists a partition π' of P' and a bijection b between P and π' such that, for all $x \in P$, (1) x is an `addRule` permission rule and $b(x)$ contains only x and the two rules produced by applying the `simAddRule` transformation to x , or (2) x is not an `addRule` permission rule, and x is an instance under some ground substitution θ of the rule pattern in some `addRule` permission rule R

of the form $\text{permit}(U, \text{addRule}(L :- \vec{L}_1)) :- \vec{L}_2$ in P , and $b(x)$ contains only the fact $\text{aux}_R(\vec{X})\theta$, where $\vec{X} = \text{vars}(L :- \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$, or (3) x is not an `addRule` permission rule, and $b(x)$ contains only x . Note that this relation is not symmetric.

Let `auxAtoms` denote the set of atoms that contain an auxiliary predicate. This includes, e.g., atoms of the form $\text{permit}(\dots, \text{addFact}(\text{aux}_R(\dots)))$.

Lemma 9.4. For all policies P and P' such that $P \simeq P'$, for all atoms a not in `auxAtoms`, $a \in \llbracket P \rrbracket$ iff $a \in \llbracket P' \rrbracket$.

Proof: We do a case analysis on whether a is an atom for an intensional or extensional predicate.

case a is extensional: The definition of \simeq implies that P and P' contain the same extensional facts except for facts for auxiliary predicates. By hypothesis, $a \notin \text{auxAtoms}$, so a is not a fact for an auxiliary predicate. Therefore, $a \in \llbracket P \rrbracket$ iff $a \in \llbracket P' \rrbracket$.

case a is intensional: Let π' and b be the partition and bijection required to exist by the definition of \simeq . First we consider the forward direction of the “iff”, i.e., we assume $a \in \llbracket P \rrbracket$ and show $a \in \llbracket P' \rrbracket$. Let D be a derivation of a using facts and rules in P . We construct a derivation D' of a using rules and facts in P' by starting with D and making the following replacements. Consider a rule x used in D . We do a case analysis based on which disjunct in the definition of \simeq holds for x . If disjunct (1) or (3) holds, then P' also contains x , so no replacement is needed. If disjunct (2) holds, for some `addRule` permission rule R in P , ground substitution θ , and fact $\text{aux}_R(\vec{X})\theta$, then the definition of \simeq implies that P' contains the two rules generated from R by the `simAddRule` transformation, and we replace x with a use of the first of those rules, instantiated with substitution θ (note that the premises of that rule are the same as the premises of R , plus one additional premise involving an auxiliary predicate, which is satisfied by the fact $\text{aux}_R(\vec{X})\theta$ in P'). It is straightforward to show that D' is a derivation of a using facts and rules in P' .

Now we consider the reverse direction of the “iff”, i.e., we assume $a \in \llbracket P' \rrbracket$ and show $a \in \llbracket P \rrbracket$. Let D' be a derivation of a using facts and rules in P' . We construct a derivation D of a using rules and facts in P by starting with D' and making the following replacements. Consider a rule x' used in D' . x' is a member of a unique set S in π' that is related to a unique element x of P by the bijection b . We replace x' with x (note that x' and x might be the same).

It is straightforward to show that D is a derivation of a using facts and rules in P . \square

The following theorem implies that \simeq is a bisimulation (for a definition of bisimulation, see, e.g., Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*, MIT Press, 1999).

Lemma 9.5. For every policy P , policy P' , and user u , if $P \simeq P'$ then (a) for every policy P_1 and operation op such that $\langle P, u:op, P_1 \rangle \in T_{\downarrow \text{aR}, -\text{rR}}$, there exists a policy P'_1 and operation op' such that $\langle P', u:op', P'_1 \rangle \in T_{-\text{aR}, -\text{rR}}$ and $P_1 \simeq P'_1$, and (b) for every policy P'_1 and operation op' such that $\langle P', u:op', P'_1 \rangle \in T_{-\text{aR}, -\text{rR}}$, there exists a policy P_1 and operation op such that $\langle P, u:op, P_1 \rangle \in T_{\downarrow \text{aR}, -\text{rR}}$ and $P_1 \simeq P'_1$.

Proof:

(a) Note that $\langle P, u:op, P_1 \rangle \in T_{\downarrow \text{aR}, -\text{rR}}$ implies that $\text{permit}(u, op) \in \llbracket P \rrbracket$. We perform a case analysis on the kind of administrative operation that op is.

case op is `removeRule`: $T_{\downarrow \text{aR}, -\text{rR}}$ does not contain `removeRule` transitions, so this case cannot occur.

case op is `addFact` or `removeFact` for a non-auxiliary predicate: In this case, $\text{permit}(u, op)$ is not in `auxAtoms`, so $\text{permit}(u, op) \in \llbracket P \rrbracket$ and $P \simeq P'$ imply, using Lemma 9.4, that $\text{permit}(u, op) \in \llbracket P' \rrbracket$, so we take op' to be the same as op , and P'_1 to be the policy obtained by executing op' in P' . It is easy to show that $\langle P', u:op', P'_1 \rangle \in T_{-\text{aR}, -\text{rR}}$ and $P_1 \simeq P'_1$.

case op is `addFact` or `removeFact` for an auxiliary predicate: The definition of \simeq implies that P does not contain `addFact` rules involving auxiliary predicates, so this case cannot occur.

case op is `addRule`: $\text{permit}(u, op) \in \llbracket P \rrbracket$ implies there is an `addRule` permission rule R in P of the form $\text{permit}(U, \text{addRule}(L :- \vec{L}_1)) :- L_2$ used with a ground substitution θ to derive $\text{permit}(u, op)$ in P ; thus, $u = U\theta$, and $op = \text{addRule}(L :- \vec{L}_1)\theta$. The definition of \simeq , and the definition of `simAddRule`, together imply that P' contains the rule $L :- \vec{L}_1, \text{aux}_R(\vec{X})$ and the rule $\text{permit}(U, \text{addFact}(\text{aux}_R(\vec{X}))) :- \vec{L}_2$, where $\vec{X} = \text{vars}(L :- \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$. Let R' denote the latter rule. We take op' to be $\text{addFact}(\text{aux}_R(\vec{X}))\theta$, and P'_1 to be the policy obtained by executing op' in P' . To see that $\text{permit}(u, op') \in \llbracket P' \rrbracket$, and hence $\langle P', u:op', P'_1 \rangle \in T_{-\text{aR}, -\text{rR}}$, note that the premises of R' are the same as the

premises of R , and they cannot be atoms for auxiliary predicates (because P does not contain occurrences of auxiliary predicates), and these premises are derivable in P , so Lemma 9.4 implies that they are derivable in P' . It is straightforward to show that $P_1 \simeq P'_1$.

(b) Note that $\langle P', u:op', P'_1 \rangle \in T_{\text{-aR,-rR}}$ implies that $\text{permit}(u, op') \in \llbracket P' \rrbracket$. We perform a case analysis on the kind of administrative operation that op' is.

case op is `addRule` or `removeRule`: $T_{\text{-aR,-rR}}$ does not contain `addRule` or `removeRule` transitions, so this case cannot occur.

case op' is `addFact` or `removeFact` for a non-auxiliary predicate: In this case, $\text{permit}(u, op')$ is not in `auxAtoms`, so $\text{permit}(u, op) \in \llbracket P' \rrbracket$ and $P \simeq P'$ imply, using Lemma 9.4, that $\text{permit}(u, op) \in \llbracket P \rrbracket$, so we take op to be the same as op' , and P_1 to be the policy obtained by executing op in P . It is easy to show that $\langle P, u:op' P'_1 \rangle \in T_{\text{-aR,-rR}}$ and $P_1 \simeq P'_1$.

case op' is `addFact` for an auxiliary predicate: $\text{permit}(u, op) \in \llbracket P' \rrbracket$ implies there is an `addFact` permission rule R' of the form $\text{permit}(U, \text{addFact}(\text{aux}_R(\vec{X}))) :- \vec{L}_2$ used with a ground substitution θ to derive $\text{permit}(u, op')$ in P' ; thus, $u = U\theta$, and $op' = \text{addFact}(\text{aux}_R(\vec{X}))\theta$. The definition of \simeq , and the definition of `simAddRule`, together imply that P' also contains the rule $L :- \vec{L}_1, \text{aux}_R(\vec{X})$, where $\vec{X} = \text{vars}(L :- \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$, and P contains the `addRule` permission rule $\text{permit}(U, \text{addRule}(L :- \vec{L}_1)) :- L_2$. Let R denote the latter rule. We take op to be $\text{addRule}(L :- \vec{L}_1)\theta$, and P_1 to be the policy obtained by executing op in P . To see that $\text{permit}(u, op) \in \llbracket P \rrbracket$, and hence $\langle P, u : op, P_1 \rangle \in T_{\text{-aR,-rR}}$, note that the premises of R are the same as the premises of R' , and they cannot be atoms for auxiliary predicates (because P does not contain occurrences of auxiliary predicates, and the definition of \simeq implies that R' was produced by the `simAddRule` transformation applied to R), and these premises are derivable in P , so Lemma 9.4 implies that they are derivable in P' . It is straightforward to show that $P_1 \simeq P'_1$.

case op' is `removeFact` for an auxiliary predicate: The definition of \simeq implies that P' does not contain `removeFact` rules for auxiliary predicates, so this case cannot

occur. □

Theorem 9.6. For every policy P_0 , set U_0 of users, and atom a not in `auxAtoms`, $\text{SG}(P_0, U_0, T_{\text{-aR,-rR}})$ contains a policy P with $a \in \llbracket P \rrbracket$ iff $\text{SG}(\text{simAddRule}(P_0), U_0, T_{\text{-aR,-rR}})$ contains a policy P' with $a \in \llbracket P' \rrbracket$.

Proof: First we prove the forward direction of the “iff”. Let P_0, \dots, P_n be a path in $\text{SG}(P_0, U_0, T_{\text{-aR,-rR}})$ such that $a \in \llbracket P_n \rrbracket$. We show by induction that $\text{SG}(\text{simAddRule}(P_0), U_0, T_{\text{-aR,-rR}})$ contains a path P'_0, \dots, P'_n from `simAddRule`(P_0) such that $P_i \simeq P'_i$. For the base case, note that the definitions of `simAddRule` and \simeq imply that $P_0 \simeq \text{simAddRule}(P_0)$. The induction step follows immediately from Lemma 9.5(a). Since $a \in \llbracket P_n \rrbracket$ and $P_n \simeq P'_n$ and $a \notin \text{auxAtoms}$, Lemma 9.4 implies $a \in \llbracket P'_n \rrbracket$.

The proof for the backward direction of the “iff” is the same except that the induction step uses Lemma 9.5(b). □

The following theorem expresses overall correctness of phase 1. In order to express the theorem cleanly, we extend abductive atom-reachability queries with an optional fifth component, which specifies the transition relation to use when constructing the state graph in the semantics of queries. When this component is omitted, the default is the full transition relation T defined in Section 2.4.

Theorem 9.7. Let $Q = \langle P_0, U_0, A, G_0 \rangle$ be an abductive atom-reachability query. Let $Q' = \langle \text{simAddRule}(P_0), U_0, A, G_0, T_{\text{-aR,-rR}} \rangle$. Then S is a comprehensive solution for Q iff S' is a comprehensive solution for Q' , where S is obtained from S' by replacing `addFact` operations for auxiliary predicates with the corresponding `addRule` operations in the plan.

Proof: Theorems 9.3 and 9.6 together imply that for every policy P_0 , set U_0 of users, and atom a not in `auxAtoms`, $\text{SG}(P_0, U_0, T)$ contains a policy P with $a \in \llbracket P \rrbracket$ iff $\text{SG}(\text{simAddRule}(P_0), U_0, T_{\text{-aR,-rR}})$ contains a policy P' with $a \in \llbracket P' \rrbracket$, and that the administrative actions in plans for reaching those policies are the same except (cf. the proof of Lemma 9.5) that plans in the former contain `addRule` operations in positions corresponding to `addFact` operations for auxiliary predicates in plans in the latter. This directly implies the desired relationship between the solutions of Q and Q' . □

Correctness of Phases 2–4: The following theorem expresses soundness of the solution produced by phases 2–4.

Theorem 9.8. Suppose the algorithm terminates, and phase 4 returns a set *solutions* of solutions. Let $\langle G, \Delta, \pi, D \rangle$ be an element of *solutions*. For every ground substitution θ such that $\text{satisfiable}(D\theta)$, the

sequence of actions $\pi\theta$ can be executed starting from $\text{simAddRule}(P_0) \cup \Delta\theta$ and leads to a policy P such that $G\theta \in \llbracket P \rrbracket$.

Proof: The proof is by induction on π . Let π_i denote the i 'th element of π , indexed starting with 0. We prove by induction that, for $0 \leq i < |\pi - 1|$, $\pi_i\theta$ is executable in P_i , i.e., there exists a (unique) policy P_{i+1} such that $\langle P_i, \pi_i\theta, P_{i+1} \rangle \in T$. Let n_g be the answer node that led to the inclusion of $\langle G, \Delta, \pi, D \rangle$ in *solutions* in phase 4. Let n_i be the admin node in $\text{proof}(n_g)$ such that $\pi_i = \text{nodeToAction}(n_i)$.

Base case: The admin node n_0 corresponding to action π_0 is a leaf in the proof graph for n_g (to see this, note that, if n_0 had any descendants in the proof graph, dependency orderings would force the corresponding actions to precede π_0 in π) and therefore depends only on facts and rules in the initial policy $\text{simAddRule}(P_0) \cup \Delta$. Note that the consistency constraint d_{init} in Figure 8 ensures that negative preconditions in the rules used to derive n_0 from facts in the initial policy are satisfied. Therefore, permission for some user in U_0 to perform $\pi_0\theta$ is derivable in the policy $\text{simAddRule}(P_0) \cup \Delta\theta$.

Step case: The tabling algorithm ensures that n_{i+1} can be derived using some subset Admin_i of the admin nodes in $\{n_0, \dots, n_i\}$, some set Abd_i of answer nodes that represent abduction, and some set of other nodes that represent derivations using inference rules. We argue that each of these three types of nodes is valid in the derivation of n_{i+1} in P_{i+1} . Regarding the admin nodes in Admin_i , the induction hypothesis implies that each node n_j in Admin_i can be derived in P_j , and that the ordering constraints defined in Figure 9 ensure that the derived atom (i.e., the partial answer) associated with n_j is not falsified by (i.e., is still derivable after) the administrative actions associated with n_{j+1}, \dots, n_i . Regarding the answer nodes in Abd_i that represent abduction, the tabling algorithm ensures that the abduced atom (i.e., the partial answer) associated with each node n in Abd_i is included in Δ , and the ordering constraints in $O_{\text{add-init}}$ and $O_{\text{rm-init}}$ defined in Figure 9 ensure that the abduced atom is not falsified by the administrative actions associated with admin nodes n_0, \dots, n_i . Regarding the other nodes that represent derivations using inference rules, note that, since phase 1 eliminates addition and removal of rules, the inference rules do not change and are valid for use in derivations in any step (i.e., after any sequence of administrative operations). Note that the consistency constraint d_{init} and the ordering constraints in $O_{\text{add-init}}$ and $O_{\text{add-rmvd}}$ ensure that negative preconditions in these inference rules are satisfied.

Finally, note that $G\theta$ is derivable from the last policy $P_{|\pi|}$ in this sequence, because $n_{|\pi|}$ is the answer node n_g (the dependence constraints ensure that n_g appears last in the sequence), hence the partial answer

associated with $n_{|\pi|}$ is G . \square

The following theorem expresses comprehensiveness of the solution produced by phases 2–4.

Theorem 9.9. Let $Q' = \langle \text{simAddRule}(P_0), U_0, A, G_0, T_{\text{-aR, -rR}} \rangle$ be an abductive atom-reachability query. Suppose the algorithm terminates and phase 4 returns a set *solutions* of solutions. Suppose there exists a set Δ_m of ground facts, and a ground instance G_m of G_0 , such that a policy that derives G_m is reachable in $\text{SG}(\text{simAddRule}(P_0) \cup \Delta_m, U_0, T_{\text{-aR, -rR}})$ via a path that does not involve repeated administrative operations. Let π_m be the sequence of administrative actions labeling the edges of such a path. Suppose also that Δ_m is a minimal residue for G_m , i.e., for all $\Delta \subset \Delta_m$, G_m is not reachable in $\text{SG}(\text{simAddRule}(P_0) \cup \Delta, U_0, T_{\text{-aR, -rR}})$. Then there exists $\langle G, \Delta, \pi, D \rangle \in \text{solutions}$ such that for some ground substitution θ , $\Delta\theta = \Delta_m$ and $G\theta = G_m$ and $D\theta = \text{true}$.

Proof: Our proof of comprehensiveness is based on the completeness theorem for Becker *et al.*'s tabling algorithm, which we described at the end of Section 4.3.1. As discussed in Section 4.3, replacing the subsumption check in `processAnswer` with an α -equality check is needed for comprehensiveness, and inserting the call to `noCyclicDeriv` to eliminate cyclic derivations improves termination without compromising comprehensiveness.

Recall that the `simAddRmFact` transformation in phase 2 changes the semantics of the policy such that an atom a that may be added by an active administrator is derivable through the new inference rules, and an atom a that may be removed by an active administrator is “removed” (in the sense that $!a$ is derivable), even though the administrators do not necessarily add or remove them, and in particular, even though those actions might not appear in plans that lead to the goal. This change to the semantics of the policy does not cause phase 3 to overlook any possible derivations of the goal. To see this, recall that phase 3 does not interpret the negation symbol as negation (recall that it treats the negation symbol as part of the predicate name) and hence may construct proof graphs corresponding to plans that lead to policies containing both an atom and its negation. In effect, the transformed policy used in phase 3 does not contain negation and hence is monotonic, so increasing the set of derivable facts allows more derivations to be considered and cannot cause any derivation to be overlooked. Finally, note that the constraints in phase 4 do not eliminate representations of minimal-residue solutions, because they express only the conditions needed to ensure that candidate solutions are feasible. \square

Overall Correctness: Next we state the correctness of the overall algorithm. Let $\llbracket \langle G, \Delta, \pi, D \rangle \rrbracket =$

$\{\langle G\theta, \pi, \Delta\theta \mid \text{ground}(\theta) \wedge D\theta = \text{true} \rangle\}$.

Theorem 9.10. Let $Q = \langle P_0, U_0, A, G_0 \rangle$ be an abductive atom-reachability query. Suppose the algorithm terminates and phase 4 returns a set *solutions* of solutions. Then *solutions* is a comprehensive solution to Q .

Proof: This follows from the correctness of phase 1, as expressed in Theorem 9.7, and the soundness and comprehensiveness of phases 2–4, as expressed in Theorems 9.8 and 9.9, respectively. \square

10 EXAMPLE PROOF GRAPHS

Figures 6 and 7 present proofs ψ_1 and ψ_2 generated for the example query in Section 3.1. The tables in the figures list the index and residue for each node; the partial answer equals the index for each node in these figures. In both figures, node n_g is an answer node for the goal G_0 . A directed edge $\langle n, n' \rangle$ in a proof graph indicates that n is an answer node for a subgoal resulting from the rule used to derive node n' . For example, in Figure 7, n_7 derives `hasAct(hp01, pOfc(gwHosp))`, which is used in n_6 to derive `aux2.5()`; in the context of the original policy, this corresponds to a use of `addRule` permission rule (2.5) in Figure 2 to add a rule defining membership in the `trCli` role. That added rule is used in node n_1 to derive `clil's` membership in `trCli(pat, gwHosp)`; the other children of n_1 represent derivations of the premises of that rule.

11 PSEUDOCODE FOR PHASE 4

Pseudocode for phase 4 appears in Figures 8, 9, and 10. `isAdmin(n)` holds iff n is an administrative node (also called an “admin node”, for short). `isAddFact(n)` holds iff `rule(n)` is a transformed `addFact` permission rule. `isRmFact(n)` holds iff `rule(n)` is a transformed `removeFact` permission rule. For a Boolean expression O in disjunctive normal form (DNF), `disjuncts(O)` returns the set of disjuncts (each of which is a conjunction) in O . Disequalities may contain wildcards; specifically, disequalities may have the form $_ \neq t$, where t does not contain wildcards (because wildcards cannot appear as arguments of constructors). The function `satisfiable(d)` tests satisfiability of a disequality d . It handles wildcards as follows: a disequality of the form $_ \neq t$ is not satisfiable.

Example: Figures 11 and 12 show the ordering constraints produced by the `orderingConstraints` function in Figure 9 for node n_g in the proof graphs ψ_1 and ψ_2 in Figures 6 and 7, respectively.

Removal of subsumed solutions: Another consequence of replacing the subsumption check in `processAnswer` with an equality check is that phase 3 may produce solutions subsumed by other solutions. Informally, a solution S is subsumed by a solution S' if S represents fewer ground solutions

$$\begin{aligned}
 O_{\psi_1} &= C_1 \vee C_2 \vee C_3 \vee C_4 \\
 C_1 &= \langle n_2, n_3, f \rangle \wedge \langle n_3, n_2, f \rangle \wedge \langle n_2, n_g, f \rangle \wedge \langle n_4, n_g, f \rangle \wedge \\
 &\quad \langle n_3, n_g, f \rangle \wedge \langle n_8, n_2, f \rangle \wedge \langle n_6, n_3, f \rangle \\
 C_2 &= \langle n_2, n_3, f \rangle \wedge \langle n_g, n_3, f \rangle \wedge \langle n_2, n_g, f \rangle \wedge \langle n_4, n_g, f \rangle \wedge \\
 &\quad \langle n_3, n_g, f \rangle \wedge \langle n_8, n_2, f \rangle \wedge \langle n_6, n_3, f \rangle \\
 C_3 &= \langle n_g, n_2, f \rangle \wedge \langle n_3, n_2, f \rangle \wedge \langle n_2, n_g, f \rangle \wedge \langle n_4, n_g, f \rangle \wedge \\
 &\quad \langle n_3, n_g, f \rangle \wedge \langle n_8, n_2, f \rangle \wedge \langle n_6, n_3, f \rangle \\
 C_4 &= \langle n_g, n_2, f \rangle \wedge \langle n_g, n_3, f \rangle \wedge \langle n_2, n_g, f \rangle \wedge \langle n_4, n_g, f \rangle \wedge \\
 &\quad \langle n_3, n_g, f \rangle \wedge \langle n_8, n_2, f \rangle \wedge \langle n_6, n_3, f \rangle
 \end{aligned}$$

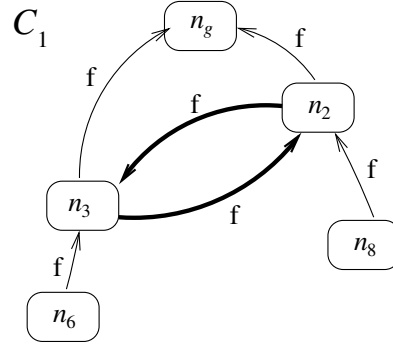


Fig. 11. Ordering constraint O_{ψ_1} generated by function `orderingConstraints` for node n_g in the proof graph ψ_1 in Figure 6, and disjunct C_1 in O_{ψ_1} represented as a labeled graph, with the unsatisfiable cycle highlighted by thicker lines. “f” abbreviates “false”.

$$\begin{aligned}
 O_{\psi_2} &= C_1 \\
 C_1 &= \langle n_6, n_g, f \rangle \wedge \langle n_2, n_g, f \rangle \wedge \langle n_9, n_2, f \rangle
 \end{aligned}$$

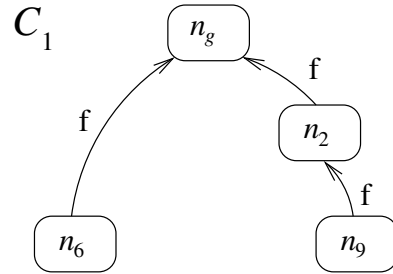
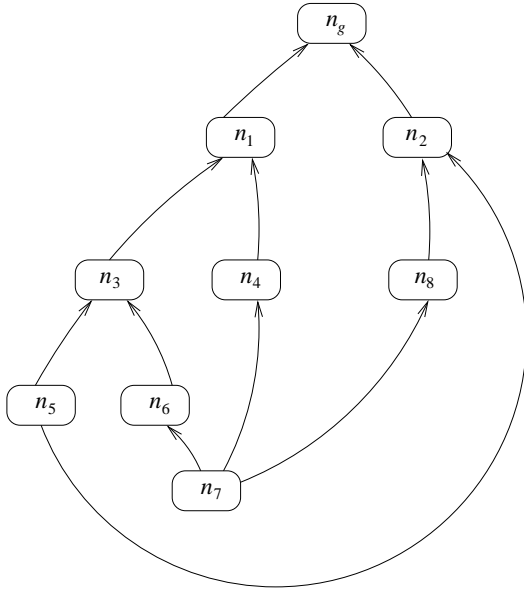


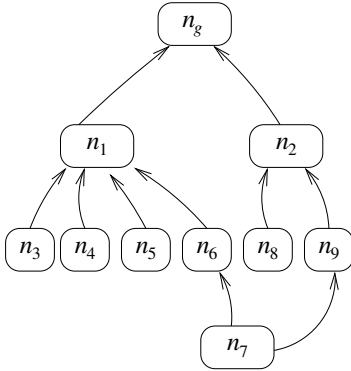
Fig. 12. Ordering constraint O_{ψ_2} generated by function `orderingConstraints` for node n_g in the proof graph ψ_2 in Figure 7, and disjunct C_1 in O_{ψ_2} represented as a labeled graph. “f” abbreviates “false”.

than S' or ground solutions with larger residues. Formally, a solution $\langle G, \Delta, \pi, D \rangle$ is subsumed by a solution $\langle G', \Delta', \pi', D' \rangle$, denoted $\langle G, \Delta, \pi, D \rangle \preceq_S \langle G', \Delta', \pi', D' \rangle$, if $|\Delta| \geq |\Delta'|$ and there exists a substitution θ such that $G = G'\theta$ and $\Delta \supseteq \Delta'\theta$ and $D \Rightarrow D'\theta$. This definition is based on the definition of the subsumption relation \preceq_A in [10, Subsection 2.4], extended to compare the tuple disequalities. Including subsumed solutions in the answer is undesirable, so the last line of pseudo-code in Figure 8 removes them.



n_g	index = <code>treatingWithoutConsent(pat1, cli1)</code> residue = {}
n_1	index = <code>memberOf(cli1, trCli(pat1, gwHosp))</code> residue = {}
n_2	index = <code>!consentTT(pat1, cli1, gwHosp)</code> residue = {}
n_3	index = <code>consentTT(pat1, cli1, gwHosp)</code> residue = {}
n_4	index = <code>aux2.4()</code> , residue = {}
n_5	index = <code>hasAct(pat1, patient)</code> , residue = {}
n_6	index = <code>aux2.6()</code> , residue = {}
n_7	index = <code>hasAct(hpol, pOfc(gwHosp))</code> residue = {}
n_8	index = <code>aux2.7()</code> , residue = {}

Fig. 6. A proof graph ψ_1 generated in phase 3 for the example query in Section 3.1 using the transformed policy $\text{simAddRmFact}(\text{simAddRule}(P_0), U_0)$ in Figure 4.



n_g	index = <code>treatingWithoutConsent(pat1, cli1)</code> residue = { <code>memberOf(cli1, wkgp(W, gwHosp, Spcty, WT))</code> , <code>encounter(EncID, pat1, W, gwHosp, Type)</code> }
n_1	index = <code>memberOf(cli1, trCli(pat1, gwHosp))</code> residue = { <code>memberOf(cli1, wkgp(W, gwHosp, Spcty, WT))</code> , <code>encounter(EncID, pat1, W, gwHosp, Type)</code> }
n_2	index = <code>!consentTT(pat1, cli1, gwHosp)</code> , residue = {}
n_3	index = <code>hasAct(cli1, cli(gwHosp, Spcty))</code> , residue = {}
n_4	index = <code>memberOf(cli1, wkgp(W, gwHosp, Spcty, WT))</code> residue = { <code>memberOf(cli1, wkgp(W, gwHosp, Spcty, WT))</code> }
n_5	index = <code>encounter(EncID, pat1, W, gwHosp, Type)</code> residue = { <code>encounter(EncID, pat1, W, gwHosp, Type)</code> }
n_6	index = <code>aux2.5()</code> , residue = {}
n_7	index = <code>hasAct(hpol, pOfc(gwHosp))</code> , residue = {}
n_8	index = <code>hasAct(pat1, patient)</code> , residue = {}
n_9	index = <code>aux2.7()</code> , residue = {}

Fig. 7. A proof graph ψ_2 generated in phase 3 for the example query in Section 3.1 using the transformed policy $\text{simAddRmFact}(\text{simAddRule}(P_0), U_0)$ in Figure 4.

12 RELATIVE EXPRESSIVENESS OF ACAR AND ARBAC97

ACAR is more expressive than ARBAC97 in many ways but is also less expressive in some ways. To see this, we consider how to express ARBAC97 in ACAR, by introducing extensional predicates that capture the policy state (primarily the set of roles, and the transitive reduction of the role hierarchy) and introducing permission rules for adding and removing facts for these predicates. These permission rules express ARBAC97's preconditions on those administrative operations. All aspects of ARBAC97's three components (URA97, PRA97, and RRA97) can

be expressed in ACAR except (1) negative preconditions for assignment of users and permissions to roles (they can be expressed if roles mentioned in negative preconditions are not junior to other roles in the role hierarchy), (2) the requirement that new role hierarchy edges can be added only between roles that are incomparable in the current role hierarchy (however, this requirement is not essential), and (3) the requirement that encapsulation of authority ranges is not violated when adding an edge to the role hierarchy (however, a reasonable conservative approximation of this requirement can be expressed in ACAR). These three exceptions have a common underlying cause: they would require applying negation to an intensional

```

solutions =  $\emptyset$ 
for  $n_g$  in  $Ans(G)$ 
  // consistency constraint for initial state: positive literals must be distinct from negative literals.
   $d_{init} = \bigwedge \{args(a) \neq args(b) \mid a \in facts(P_0) \cup residue(n_g) \wedge !b \in residue(n_g) \wedge unifiable(a, b)\}$ 
  // consistency constraint for residue: residue must be disjoint from  $nAb$ .
   $d_{nAb} = \bigwedge \{args(a) \neq args(b) \mid a \in residue(n_g) \wedge b \in nAb \wedge unifiable(a, b)\}$ 
   $d_0 = d_{init} \wedge d_{nAb}$ 
  if  $\neg$ satisfiable( $d_0$ )
    continue
   $O = orderingConstraints(n_g)$ 
  if ( $\exists c \in disjuncts(O)$ . the ordering constraints in  $c$  are acyclic)
    // the ordering constraints for  $n_g$  are satisfiable without imposing disequalities.
    // intersect residue with  $Atom_{ex}$  (the extensional atoms) to remove negative literals.
    solutions = solutions  $\cup$  { $\langle pAns(n_g), residue(n_g) \cap Atom_{ex}, plan(n_g, c), d_0 \rangle$ }
  else
    // the ordering constraints for  $n_g$  are not satisfiable in general, but might be satisfiable if disequalities are
    // imposed to ensure that some administrative operations operate on distinct atoms and therefore commute.
    for  $c$  in  $disjuncts(O)$ 
      if mightNeedRepeatedOp( $n_g, c$ )
        // See text for description of how to extend the algorithm to support repeated operations.
        return "repeated operations might be needed"
       $D_{ord} = \emptyset$ 
      //  $c$  is a conjunction (treated as a set) of labeled ordering constraints. Remove some ordering constraints  $F$ 
      // from  $c$  to make the remaining ordering constraints acyclic, and insert in  $D_{ord}$  the conjunction  $d$  of  $d_0$ 
      // and the disequalities labeling the removed ordering constraints, if  $d$  is satisfiable and not subsumed by
      // an existing element of  $D_{ord}$ . We use the algorithm in [14] to compute  $Cyc$ .
       $Cyc =$  set of all cycles in ordering constraints for disjunct  $c$ 
       $FAS = \{F \mid F \text{ contains one edge selected from each cycle in } Cyc\}$ 
      //  $smFAS$  is the set of  $\subseteq$ -minimal feedback arc sets (FASs) for disjunct  $c$ 
       $smFAS = \{F \in FAS \mid \nexists F' \in FAS. F' \subset F\}$ 
      for  $F$  in  $smFAS$ 
         $d = d_0 \wedge \bigwedge \{d' \mid \langle n_1, n_2, d' \rangle \in F\}$ 
        if satisfiable( $d$ )  $\wedge \neg(\exists \langle d', F' \rangle \in D_{ord}. d' \subseteq d)$ 
           $D_{ord} = D_{ord} \cup \{\langle d, F \rangle\}$ 
          solutions = solutions  $\cup$  { $\langle pAns(n_g), residue(n_g) \cap Atom_{ex}, plan(n_g, c \setminus F), d \rangle \mid \langle d, F \rangle \in D_{ord}$ }
      // return solutions that are not subsumed by other solutions.
    return { $s \in solutions \mid \nexists s' \in solutions. s \preceq_S s'$ }

```

Fig. 8. Pseudo-code for Phase 4. $args(a)$ returns a tuple containing the arguments of atom a .

relation, in particular, the transitive closure of the role hierarchy. That transitive relation is defined by a static set of rules, i.e., there are no `addRule` or `removeRule` permission rules that allow changing the definition of that relation. ACAR could be extended to allow negation to be applied to relations defined by a static set of rules. All of ARBAC97 could be expressed in ACAR with this extension.

```

function orderingConstraints( $n_g$ )
   $\theta = \theta_{pa}(n_g)$ 
  // dependence constraint: an admin node  $n_s$  that supports  $n$  must occur before  $n$ .
   $O_{dep} = \bigwedge \{ \langle n_s, n, false \rangle \mid n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_s \in \text{adminSupport}(n) \}$ 
  // all of the constraints below are interference-freedom constraints.
  // a removeFact node  $n_r$  that removes a supporting initial fact of a node  $n$  must occur after  $n$ .
   $O_{rm-init} = \bigwedge \{ \langle n, n_r, \text{args}(a)\theta \neq \text{args}(pAns(n_r))\theta \rangle \mid$ 
     $n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_r \in \text{proof}(n_g) \wedge \text{isRmFact}(n_r)$ 
     $\wedge n \neq n_r \wedge a \in \text{supportingInitFact}(n) \wedge \text{unifiable}(!a, pAns(n_r)) \}$ 
  // an addFact node  $n_a$  that adds a fact whose negation is a supporting initial fact of a node  $n$  must occur after  $n$ .
   $O_{add-init} = \bigwedge \{ \langle n, n_a, \text{args}(a)\theta \neq \text{args}(pAns(n_a))\theta \rangle \mid$ 
     $n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_a \in \text{proof}(n_g) \wedge \text{isAddFact}(n_a)$ 
     $\wedge n \neq n_a \wedge !a \in \text{supportingInitFact}(n) \wedge \text{unifiable}(a, pAns(n_a)) \}$ 
  // an addFact node  $n_a$  that adds a supporting removed fact of a node  $n$  must occur either before the removal
  // of that fact or after  $n$ .
   $O_{add-rmvd} =$ 
   $\bigwedge \{ \langle n_a, n_r, \text{args}(pAns(n_a))\theta \neq \text{args}(pAns(n_r))\theta \rangle \vee \langle n, n_a, \text{args}(pAns(n_a))\theta \neq \text{args}(pAns(n_r))\theta \rangle \mid$ 
     $n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_r \in \text{adminSupport}(n) \wedge \text{isRmFact}(n_r)$ 
     $\wedge n_a \in \text{proof}(n_g) \wedge \text{isAddFact}(n_a) \wedge n \neq n_a \wedge \text{unifiable}(!pAns(n_a), pAns(n_r)) \}$ 
  // a removeFact node  $n_r$  that removes a supporting added fact of a node  $n$  must occur either before the addition
  // of that fact or after  $n$ .
   $O_{rm-added} =$ 
   $\bigwedge \{ \langle n_r, n_a, \text{args}(pAns(n_a))\theta \neq \text{args}(pAns(n_r))\theta \rangle \vee \langle n, n_r, \text{args}(pAns(n_a))\theta \neq \text{args}(pAns(n_r))\theta \rangle \mid$ 
     $n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_a \in \text{adminSupport}(n) \wedge \text{isAddFact}(n_a)$ 
     $\wedge n_r \in \text{proof}(n_g) \wedge \text{isRmFact}(n_r) \wedge n \neq n_r \wedge \text{unifiable}(!pAns(n_a), pAns(n_r)) \}$ 
  // conjoin all ordering constraints and convert the formula to disjunctive normal form.
   $O = \text{DNF}(O_{dep} \wedge O_{rm-init} \wedge O_{add-init} \wedge O_{add-rmvd} \wedge O_{rm-added})$ 
  // for each disjunct  $c$  of  $O$ , merge labeled ordering constraints for the same pair of nodes,
  // so the disjunct represents a graph with at most one edge between each pair of nodes.
  for  $c$  in disjuncts( $O$ )
    while  $\exists n_1, n_2, D, D'. \langle n_1, n_2, D \rangle \in c \wedge \langle n_1, n_2, D' \rangle \in c$ 
      replace  $\langle n_1, n_2, D \rangle$  and  $\langle n_1, n_2, D' \rangle$  with  $\langle n_1, n_2, D \wedge D' \rangle$  in  $c$ 
  return  $O$ 

```

```

support( $n$ ) =  $\{n' \in \text{proof}(n) \mid \text{isAns}(n') \wedge n' \neq n \wedge \neg \exists n_a. \text{isAdmin}(n_a) \wedge \text{descendant}(n, n_a) \wedge \text{descendant}(n_a, n')\}$ 
adminSupport( $n$ ) =  $\{n' \in \text{support}(n) \mid \text{isAdmin}(n')\}$ 
supportingInitFact( $n$ ) =  $\{pAns(n') \mid n' \in \text{support}(n) \wedge (\text{rule}(n') \in \text{facts}(P_0) \vee \text{rule}(n') = \text{abduction})\}$ 

```

Fig. 9. Function to generate ordering constraints for an answer node n_g , and some auxiliary functions.

```

function plan( $n_g, c$ )
   $N = \{n \in \text{proof}(n_g) \mid \text{isAdmin}(n)\}$ 
   $\pi =$  a permutation of  $N$  such that
     $\forall n, n' \in N. \langle n, n', \_ \rangle \in c \Rightarrow n$  precedes  $n'$  in  $\pi$ 
  return map(nodeToAction,  $\pi$ )

function nodeToAction( $n$ )
  if isAddFact( $n$ )
     $op = \text{addFact}(pAns(n))$ 
  else
     $op = \text{removeFact}(pAns(n))$ 
     $n' =$  the node in children( $n$ ) with partial
      answer of the form  $u0(\dots)$ 
     $u =$  the argument of  $u0$  in  $pAns(n')$ 
  return  $u:op$ 

```

Fig. 10. Functions used for plan construction. $\text{plan}(n_g, c)$ produces a plan from an answer node n_g and a conjunction c of ordering constraints; it calls $\text{map}(f, s)$, which maps function f along sequence s and returns the sequence containing the results. $\text{nodeToAction}(n)$ computes the administrative action corresponding to an administrative answer node n .