Verification of Security Policy Administration and Enforcement in Enterprise Systems

A Dissertation Presented

by

Puneet Gupta

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2011

Stony Brook University

The Graduate School

Puneet Gupta

We, the dissertation committee for the above candidate for

the degree of Doctor of Philosophy,

hereby recommend the acceptance of this dissertation.

Scott D. Stoller – Dissertation Advisor

Professor, Computer Science Department

R. Sekar – Chairperson of Defense

Professor, Computer Science Department

C. R. Ramakrishnan – Committee Member

Associate Professor, Computer Science Department

Jorge Lobo – External Committee Member

Research Staff Member, Thomas J. Watson Research Center, Hawthorne

This dissertation is accepted by the Graduate School.

Lawrence Martin

Dean of the Graduate School

Abstract of the Dissertation

Verification of Security Policy Administration and Enforcement in Enterprise Systems

by
Puneet Gupta

Doctor of Philosophy
in
Computer Science
Stony Brook University
2011

The scale and complexity of security policies in enterprise systems makes it difficult to ensure that they achieve higher-level security goals. This dissertation explores two important ways in which policy analysis can help: reachability analysis for administrative policies, and analysis of policy enforcement in enterprise systems.

An administrative policy specifies how each user in an enterprise may change the policy. Fully understanding the consequences of an administrative policy can be difficult, 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 dissertation presents a rule-based access control policy language, a rule-based administrative policy model that controls addition and removal of rules and facts, 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.

Many security requirements for enterprise systems can be expressed in a natural way as high-level access control policies, but are not enforced by a single mechanism that directly interprets such policies. A high-level policy may refer to abstract information resources, independent of where the information is stored; it controls both direct and indirect accesses to the information; it may refer to the context of a request, i.e., the request's path through the system; and its enforcement point and enforcement mechanism may be unspecified. Enforcement of a high-level policy may depend on the system architecture and the configurations of a variety of security mechanisms, such as firewalls, database access control, and application-level access control. This dissertation presents a framework for expressing high-level policies, a method for verifying that a high-level policy is enforced, and an algorithm for determining a trusted computing base for each resource.

Contents

List of Figures

Acknowledgements

"Feeling gratitude and not expressing it is like wrapping a present and not giving it."

– William Arthur Ward

First and foremost I'd like to thank my advisor Prof. Scott D. Stoller who taught, trained and guided me through the last five years. I am extremely grateful for his support and confidence in me, and his patience in persistently teaching and training me to accomplish this feat. In research, he taught me how to coagulate my thoughts into problems, how to approach problems, how to present my solutions, and to critically examine my work to improve it. Personally, he taught me, through example, to maintain patience and perseverance towards achieving targets in life. Through himself, he set a standard by which I will always try to live.

Every journey has a beginning and every academic pursuit begins with a first teacher. In my case, I couldn't have asked for a better first teacher in my life – my brother Amit Gupta – who constantly taught me and encouraged me since I was a child; being impartial in criticizing me for my mistakes and at the same time being more than just a teacher in helping me correct them. I feel immense pride and gratitude in having him as a brother. His constant encouragement and guidance throughout my life shaped me into who I am.

A family defines a person as it lays the foundation of his character and teaches him the values to live by. In that respect I am eternally grateful for my parents, Banshi Dhar Gupta and Uma Gupta, who taught me the values and principles I live by. I'd also like to thank my sister-in-law, Nanditha Gupta, who advised me on and helped me address most of my day-to-day troubles so that I can better focus on my research and graduate studies.

Friends have always been one of the most significant aspect of my life. I have been blessed with the best friends one could ask for. Sushma, Umang, Gyanit, Sadhika, Lalit, Komal, Sejal, Tejo, Vaishali, Srivani, Tapsie, Riccardo – I am thankful for having you as my friends for you have been more than just friends, and have treated me as family and provided me with support and encouragement whenever I needed it.

I'd like to thank Prof. R. Sekar and Prof. C. R. Ramakrishnan for their suggestions and guidance that helped shape this dissertation and, also, my research in this area. Without their valuable feedback, this would not have been possible. I'd also like to thank Dr. Jorge Lobo for taking an interest in this work and providing feedback on this thesis which helped improve it further.

Last, but not the least, one cannot achieve such a milestone without support from his colleagues and friends at workplace. Whether discussing research or helping me figure out any personal matters, I am glad I had my labmates to spend time with. I thank you all in chronological order of meeting: Michael Gorbovitski, Tuncay Tekle, Jon Brandvein, Bo Lin, Zhongyuan Xu.

Chapter 1

Introduction

1.1 Research Motivation

One of the most important problems for information systems is controlling access to information. Allowing legitimate access to authorized users while preventing unauthorized access is the purpose of an *access control* system. Over the years many models have been proposed and implemented to achieve access control, notably Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-based Access Control (RBAC) [SCFY96b].

An access control system provides a framework for specification and enforcement of *security policies*. Since an organization's security policy must evolve, an access control framework should also define operations for modifying the policy and support policies that control the use of that operations. Such policies are called *administrative policies*. The scale and complexity of security policies, including administrative policies in large organizations make it difficult to ensure that a policy achieves intended higher-level security goals. This dissertation explores analysis techniques that help security administrators verify that security policies are correctly formulated and enforced.

Security Policy Administration: The increasingly complex security policies required by applications in large organizations cannot be expressed in a precise and robust way using access-control lists (ACLs) or role-based access control (RBAC). This motivated the development of attribute-based access control frameworks with rule-based policy languages. These frameworks allow policies to be expressed at a higher level of abstraction, making the policies more concise and easier to administer.

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. Several administrative frameworks have been proposed for role-based access control, starting with the classic ARBAC97 model [SBM99]. Fully understanding the implications of an administrative policy can be difficult, due to unanticipated interactions between interleaved sequences of changes by different users. This motivated research on analysis of administrative policies. For example, analysis algorithms for ARBAC97 and variants thereof can answer questions such as user-permission reachability, which asks whether changes by specified users can lead to a policy in which a specified user has a specified permission [LT06, SYRG07, JLT$^+$08, SYSR11, SYGR11]. Existing work on administrative frameworks for rule-based access control and analysis algorithms for such frameworks [Bec09, BN10] consider only addition and removal of facts, not rules.

Security Policy Enforcement: In large systems, enforcement of security policies is distributed, not centralized: different components may be responsible for enforcing different aspects of the organization's overall higher-level security policy. Verifying that the system architecture and the security configurations of the components together achieve global, high-level security goals is an important problem. Many security requirements for enterprise systems can be expressed in a natural way as high-level access control policies. These policies may be high-level in multiple ways. First, they may refer to abstract *information resources*, independent of where the information is stored. For example, consider the requirement that only employees in the registrar's office may access student transcripts. This should apply regardless of whether the transcripts are all stored in one DBMS, partitioned (e.g., by campus, college, or grad/undergrad) among multiple DBMSs, saved in backup files, etc. Second, a high-level policy controls both direct and *indirect* accesses to the information. For example, the above policy implies that other users cannot read transcripts by directly accessing them in a DBMS or by invoking operations of an application (possibly running with a different userid) that accesses the database and returns information from the transcripts. Third, a high-level policy may refer to the *context* of a request, i.e., the request's path through the system. For example, a policy might state that employees in the registrar's office are permitted to access student transcripts only via a web browser running on a host in the campus network and requesting the information from the Registrar Application Server. Note that this is analogous to the use of calling context (stack introspection) in the Java security model. Fourth, the policies may be *delocalized*, in the sense that the enforcement point and enforcement mechanism may be unspecified. For example, if transcripts are stored in a DBMS, the above requirement might be enforced in the DBMS or an application that connects to the DBMS. With the latter approach, the system should be designed so that unauthorized users cannot circumvent that application and access the DB directly. This policy might also be enforced in part by the operating system (based on login permissions and file permissions on the relevant servers) and the network (blocking connections to the server from hosts on which unauthorized users have login permissions).

Each high-level policy is enforced by one or more security mechanisms in a system (perhaps involving DBMSs, middleware, operating systems, file systems, firewalls, etc.). Enforcement also depends on the system architecture, which affects the possible paths that requests can take through the system. We sometimes refer to the configurations of security mechanisms as *low-level policies*. Ensuring that the low-level policies, together with a given system architecture, correctly enforce given high-level policies is a challenging problem.

Since enforcement of the high-level policies that control access to an information resource might involve multiple hardware and software components in the system, a natural question during security analysis is to identify a *trusted computing base* (TCB) for each information resource. Note that the answer may depend on the low-level policies as well as the system architecture.

1.2 Research Contribution

Verification of Security Policy Administration in Rule-based Access Control: We define a rule-based access control policy language, with a rule-based administrative framework that controls addition and removal of both facts and rules. We call this policy framework ACAR (Access Control and Administration using Rules). It allows administrative policies to be expressed concisely and at a desirable level of abstraction. Nevertheless, fully understanding the implications of a rule-based administrative policy in ACAR is, in some ways, even more difficult 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.

We present 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. The atom could be for the user-permission predicate, indicating that a user has a particular permission, or for any other predicate. To the best of our knowledge, this is the first analysis algorithm for a rule-based policy framework that considers changes to the rules as well as changes to the facts.

It is often desirable to be able to analyze rule-based policies with incomplete knowledge of the facts in the initial (i.e., current) policy. For example, a database containing those facts might not exist yet (if the policy is part of a system that has not been deployed), or it might be unavailable to the policy engineer due to confidentiality restrictions. Even if a database of facts exists and is available to the policy engineer, 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, e.g., because the policy might be deployed in other systems with different 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 under an encounter (an encounter is a record of a patient's admission to the hospital). A policy auditor might want to analyze the rules in the hospital policy to compute all administrative action plans that may allow a user to be a treating clinician for a patient, independent of specific data about patient encounters, which might be incomplete or non-existent during policy design. Further, even if there is some data on patient encounters, it is transient and a more thorough analysis should consider more general scenarios.

There are two approaches to solve such an analysis problem. In the deductive approach, the user specifies assumptions—in the form of constraints—about the initial facts, and the analysis algorithm determines whether the goal is reachable under those constraints. However, formulating appropriate constraints might be difficult, and might require multiple iterations of analysis and feedback. We adopt an abductive approach, in which the analysis algorithm determines conditions on the set of facts in the initial policy under which the given 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 and Nanz's abductive policy analysis for a rule-based policy language [BN08, BMD09], and our algorithm builds on their tabling-based policy evaluation algorithm. The main difference between their work and ours is that they analyze a fixed access control policy: they do not consider any administrative framework or any changes to the rules or facts in the access control policy. Also, they do not consider constructors or negation, while our policy language allows constructors and allows negation applied to extensional predicates.

This work is also described in the conference paper [GSX11]. The major differences are that this presentation of the abductive analysis algorithm uses the tabling algorithm in [BMD09], while [GSX11] uses the tabling algorithm in [BN08], and more discussion, examples, and correctness proofs have been added.

Verification of Security Policy Enforcement in Enterprise Systems: Our second main research contribution is a framework to verify enforcement of security requirements, which we call high-level security policies, in enterprise systems. More specifically, our contributions include (1) explicit identification of the above characteristics of high-level policies, (2) a framework that allows *convenient* and *formal* specification of such high-level policies, modeling of low-level policies, and modeling of relevant aspects of system architecture, (3) a method for verifying that the low-level policies in a system correctly enforce ("implement") the high-level policies, and (4) an algorithm for computing a trusted computing base (TCB) for a component or information resource.

Although there is a sizable literature on formal specification and analysis of security policies, we are not aware of any previous work that explicitly deals with high-level policies with these characteristics. The interplay between system architecture and the policies has a significant impact on our framework. Frameworks for security policy specification and analysis generally ignore system architecture and request context (in the sense described above), except for specialized frameworks for network (e.g., firewall) policy analysis. Although our framework is broad and flexible enough to model relevant aspects of network security and operating system security, our focus is on application-level security policies.

This work is also described in the conference paper [GS09].

1.3 Outline

The remainder of the thesis is structured as follows. Chapter 2 presents a language and a framework, called ACAR, for expressing administrative policies in a rule-based access control environment. Chapter 3 is a case study of a policy for a healthcare network used to demonstrate the practicality and applicability of this language. Chapter 4 presents a symbolic algorithm for abductive atom-reachability analysis of administrative policies in ACAR. The algorithm is evaluated on the healthcare network policy from chapter 3. Chapter 5 presents a policy framework for verification of enforcement of high-level policies in enterprise systems. When a policy is not enforced, the algorithm returns a counterexample that demonstrates the vulnerability. Chapter 6 summarizes the contributions and discusses future work.

Chapter 2

A Language for Security Policy Administration in Rule-based Access Control

This chapter defines a rule-based access control framework called ACAR (Access Control and Administration using Rules). ACAR includes a rule-based administrative framework that controls addition and removal of both facts and rules. ACAR allows administrative policies to be expressed concisely and at a desirable level of abstraction.

2.1 Running Example

As a running example, we use a fragment of the healthcare network case study presented in full in chapter 3. The running example focuses on the policy for appointing a user as a treating clinician for a patient at the `getWellHosp` within the healthcare network. The `getWellHosp` policy officer (HPO) can add rules that define membership in the `treatingClinician` role. We'll be using this example to demonstrate various aspects of the framework and the analysis problem presented in this thesis.

The following roles and predicates are used in this example.

- **Predicates:** The `permit(u, op)` means user `u` has the permission to execute operation `op`. The predicate `memberOf(u, r)` means that user `u` is a member of the role `r`. A user `u` can be actively operating under a role `r` that he is a member of if `hasActivated(u,r)` is true. For the

purpose of this example, we differentiate between role membership that is "inferred" through rules and is represented by the intensional predicate `memberOf` and role membership directly assigned to a user and is represented by the extensional predicate `directMemberOf`. The distinction between these two predicates is elaborated later in section 3.3. Note that `directMemberOf(u, op)` means `memberOf(u, op)`. Other predicates used in this example are `consentToTreatment(Pat, Cli, Fac)`, which means that clinician `Cli` has patient `Pat`'s consent to treat him at facility `Fac`, and
`encounter(EncID, Pat, Wkgp, Fac, Type)`, which means that there exists a patient encounter with unique identifier `EncID` for patient `Pat` at facility `Fac` of type `Type` and is handled by workgroup `Wkgp`.

- **Roles:** Members of the `treatingClinician(Pat, Fac)` role are users who are treating clinicians for patient `Pat` at facility `Fac`. Members of the `policyOfficer(Fac)` role are the policy officers at facility `Fac`. Members of the `clinician(Fac, Spcty)` role are clinicians at facility `Fac` under specialty `Spcty`. Members of the `workgroup(Wkgp, Fac, Spcty, WkgpType)` role are members of the workgroup `Wkgp`, which is of type `WkgpType`, under specialty `Spcty` at facility `Fac`. A workgroup is associated with a patient through the encounter predicate. Informally, an encounter relates a patient to a workgroup treating the patient for that encounter type. Policies for workgroup assignment, patient role assignment, agent appointment, clinician appointment and patient-workgroup association are presented in chapter 3. Members of the `patient` role are patients. Members of the `agent(Pat)` role are agents to the patient `Pat`.

The administrative policy allows HPO to define the `treatingClinician` role using the following two kinds of rules:

1. if the user has at least explicit consent to treatment for a patient, then he can be a treating clinician for that patient

```
(3.5.13)
permit(User, addRule(
               memberOf(Cli,
                        treatingClinician(Pat, getWellHosp))
               :- consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

or

2. 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

```
(3.5.12)
permit(User,
       addRule(memberOf(Cli, treatingClinician(Pat, getWellHosp))
               :- hasActivated(Cli, clinician(getWellHosp, Spcty)),
                  memberOf(Cli,
                           workgroup(Wkgp, getWellHosp, Spcty,
                                     WkgpType)),
                  encounter(EncID, Pat, Wkgp, getWellHosp, Type)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

In this description of the policy rules, "at least" indicates that the stated requirement is the minimal one; the HPO may impose additional requirements if desired, as discussed in more detail later in this chapter.

The HPO may add rules that allow patients and their agents to grant (rules 3.5.7, 3.5.9) and revoke (rules 3.5.8, 3.5.10) consent to treatment. Patient's are members of the `patient` role. An agent of a patient `Pat` is a member of the `agent(Pat)` role.

```
(3.5.7)
permit(User,
       addRule(
               permit(Pat,
                      addFact(consentToTreatment(Pat, Cli,
                                                 getWellHosp)))
               :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getWellHosp))

(3.5.8)
permit(User,
       addRule(
               permit(Pat,
                      removeFact(consentToTreatment(Pat, Cli,
                                                    getWellHosp)))
               :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getWellHosp))

(3.5.9)
permit(User,
       addRule(
               permit(Ag,
                      addFact(consentToTreatment(Pat, Cli,
                                                 getWellHosp)))
              :- hasActivated(Ag, agent(Pat))))
:- hasActivated(User, policyOfficer(getWellHosp))

(3.5.10)
permit(User,
       addRule(
             permit(Ag,
                  removeFact(consentToTreatment(Pat, Cli,
                                                getWellHosp)))
              :- hasActivated(Ag, agent(Pat))))
:- hasActivated(User, policyOfficer(getWellHosp))
```

The initial policy might also contain facts, such as:

```
directMemberOf(joeCool, clinician(getWellHosp, surgeon)).
directMemberOf(peppermintPatty, patient).
directMemberOf(charlieBrown, policyOfficer(getWellHosp)).
```

2.2 Policy Language

The policy language is a rule-based language with constructors (functors) and negation (denoted "`!`"). The language definition is parameterized by the set of constructors. In examples, we take that set of constructors to be the set of constructors that appear in the given problem instance. 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 by administrative operations), parameterized roles, etc. The grammar ensures that negation is applied only to extensional predicates; this is why we distinguish intensional and extensional predicates. The grammar appears below. *p_in* ranges over intensional predicates, *p_ex* ranges over extensional predicates, *c* ranges over constructors (functors), and *v* ranges over variables. The grammar is parameterized by the sets of predicates, variables, and constructors; these sets may be finite or countable. Predicates and constructors start with a lowercase letter; variables start with an uppercase letter. Constants are represented as constructors with arity zero; the empty parentheses are elided. The special symbol "_", called *wildcard*, is used only in negative preconditions. Wildcards are discussed in more detail in the **Negation** paragraph later in this section. Subscripts *in* and *ex* are mnemonic for intensional and extensional, respectively. 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 substitution θ is *ground*, denoted ground(θ), if it maps variables to ground terms. A *policy* is a set of rules and facts.

term	$::=$	$v \mid c(\mathit{term}^*)$	$\mathit{literal}$	$::=$	$\mathit{atom}_{ex} \mid \mathit{atom}_{neg} \mid \mathit{atom}_{in}$
atom_{ex}	$::=$	$p_{ex}(\mathit{term}^*)$	rule	$::=$	atom_{in} `:-` $\mathit{literal}^*$
atom_{in}	$::=$	$p_{in}(\mathit{term}^*)$	fact	$::=$	ground instance of atom_{ex}
atom_{neg}	$::=$	$!p_{ex}((\mathit{term} \mid \_\,)^*)$			

The predicate `permit(user, operation)` specifies permissions, including permissions for administrative operations, as discussed below.

An example of an intensional predicate is `memberOf(User, Role)`, since a role can be assigned directly to a user or inferred through user attributes. For example, a doctor `Doctor` can be appointed directly as a treating clinician to a patient `Pat`, represented by a fact `memberOf(Doctor, treatingClinician(Pat, getWellHosp))` or can be inferred as a treating clinician for `Pat` if he is a member of a workgroup that is treating `Pat`, represented by the rule:

```
memberOf(Doctor, treatingClinician(Pat, getWellHosp))
:- hasActivated(Doctor, clinician(getWellHosp, Spcty)),
   memberOf(Doctor, workgroup(Wkgp, getWellHosp, Spcty, WkgpType)),
   encounter(EncID, Pat, Wkgp, getWellHosp, Type)
```

2.3 Administrative Framework

The administrative framework defines an API for modifying policies. The operations in the API are `addRule`(*rule*), `removeRule`(*rule*), `addFact`(atom_{ex}), and `removeFact`(atom_{ex}). Let AdminOp = {`addRule`, `removeRule`, `addFact`, `removeFact`}. In addition, the framework defines how permission to perform these operations are controlled. These permissions are expressed using the `permit` predicate but given a special interpretation, as described below.

A *permission rule* is a rule whose conclusion has the form `permit(...)`. 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 $op \in$ AdminOp. In a well-formed policy, the argument to each occurrence of `addFact` and `removeFact` must be an extensional atom (it does not need to be ground).

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.

2.4 Policy Semantics

Intuitively the semantics $[\![P]\!]$ of a policy P contains all atoms deducible from P. Formally, the semantics $[\![P]\!]$ of a policy P is the least fixed-point of F_P, defined by $F_P(I) = \{a\theta \mid (a \texttt{ :- } a_1, \ldots, a_m, !b_1, \ldots, !b_n) \in P \wedge (\forall i \in [1..m] : a_i\theta \in I) \wedge (\forall i \in [1..n] : b_i\theta \notin I)\}$. To simplify notation, this definition assumes that the positive premises appear before the negative premises; this does not affect the semantics. This semantics is essentially the same as the traditional semantics of derivability for logic programs. Atoms in the semantics are ground except that arguments of `addRule` and `removeRule` may contain variables. Limiting negation to extensional predicates ensures that F_P is monotonic. By the Knaster-Tarski theorem, the least fixed point of F_P can be calculated by repeatedly applying F_P starting from the empty set. Safety of the policy implies that, during this calculation, whenever $b_i \notin I$ is evaluated, b_i is ground; this simplifies the semantics of negation. The semantics of a policy may be infinite, due to the presence of constructors, which can lead to terms with arbitrary depth in the semantics. We sometimes write $P \vdash a$ (read "P derives a") to mean $a \in [\![P]\!]$.

Also, 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.

2.5 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. We allow this exception because it 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 $\texttt{permit}(\ldots, op(\texttt{permit}(\ldots, op'(\ldots)))$ with $op \in \text{AdminOp}$ and $op' \in \text{AdminOp}$; in other words, it is a rule that permits addition and removal of administrative permission rules. 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 $op = \texttt{addRule}$ and $op' = \texttt{addFact}$. 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.

2.6 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'.

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'. These comparisons ignore renaming of variables.

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

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

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

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

We interpret addRule permission rules to allow addition of rules that are stricter than the specified rule pattern because this greatly increases flexibility for the administrator to customize the rules being added, while not allowing the administrator to add rules that violate desired safety properties of the policy. For example, a healthcare network-wide administrative policy stating that a facility's human resource (HR) manager may appoint users who have federal certification for medical practice as clinicians at that facility by making them direct (instead of inferred) members of the clinician role. This rule may be added by the getWellHosp policy officer and is network-wide.

```
permit(PO,
       addRule(
        permit(HR,
              addFact(directMemberOf(Cli, clinician(Facility, Spcty))))
        :- memberOf(HR, hrManager(Facility)),
           federalCertifiedClinician(Cli)))
:- hasActivated(PO, policyOfficer(Facility))
```

The predicate directMemberOf(User, Role) means that User has been directly assigned to Role. In our case study presented in chapter 3, the extensional predicate directMemberOf is used in the rules defining the intensional predicate memberOf (see rule 3.3.1).

Using this administrative rule, a getWellHosp policy officer might add a rule with additional premises that restrict the HR manager to appoint only clinicians who are also certified by the state. This could be a legal requirement in some states. For example, policyOfficer(getWellHosp) might add the following rule to getWellHosp policy:

```
permit(HR, addFact(directMemberOf(Cli, clinician(Facility, Spcty))))
:- memberOf(HR, hrManager(Facility)),
   federalCertifiedClinician(Cli),
   stateCertifiedClinician(Cli)
```

where stateCertifiedClinician(Cli) is a predicate defined under getWellHosp policy that means the clinician is certified by the state the getWellHosp is in. Note that the healthcare network may be a nationwide organization with facilities in different states. Each state may have its own clinician certification criteria. Hence it is more convenient for the policy officers at the facilities to incorporate their state's requirements in such policy rules. Note that the added rule is stricter than the rule pattern specified in the `addRule` permission rule.

The statement above that allowing an administrator to add stricter rules does not enable them to violate safety requirements, means, specifically, that any atom reachability goal that can be reached at all can be reached by adding rules as they appear in the `addRule` permission rules, i.e., without adding premises or instantiating variables. This observation is exploited by our analysis algorithm (see Section 4.4.). This is mainly a consequence of prohibiting negation on intensional predicates. To see why this is the case, suppose for a moment that we allow negation for intensional relations, and consider the following example:

```
user(alice).
user(bob).
user2(alice).
permit(adam, addRule(employee(X) :- user(X))) :-
permit(eve, read(data)) :- employee(alice), ! employee(bob)
```

If `adam` adds the rule "`employee(X) :- user(X)`", then `eve` does not get permission to read the data. If `adam` adds a stricter version, such as "`employee(alice) :- user(alice)`" or "`employee(X) :- user(X), user2(X)`", then `eve` gets permission to read the data.

2.7 Decentralized Policies

The healthcare network case study in chapter 3 involves a decentralized policy. We express decentralized policies using an extended form of atom, namely, *term* `issues` *atom*, where *term* represents the user that issued the statement. Examples of such rules are rules 3.5.25, 3.5.28 presented in chapter 3. We currently treat this as syntactic sugar that gets eliminated by making the issuer an argument of the predicate in the atom. In future work, we plan to extend our framework and analysis to thoroughly support trust management and trust negotiation.

2.8 Negation

The grammar limits negation to extensional predicates; allowing unrestricted use of negation would significantly complicate the semantics and the analysis algorithm. Our experience with case studies suggests that this restriction on negation is acceptable for many typical policies. To increase the expressiveness, the language allows wildcards, denoted by underscores, as arguments of an extensional predicate p (but not as arguments of constructors) in a negative literal, provided the policy does not contain `removeFact` permission rules for predicate p. Intuitively, using a wildcard as an argument in a negative premise represents a quantification over the value of that argument. Section 4.4 explains why wildcards are limited to predicates for which there are no `removeFact` permission rules.

Chapter 3

Case Study: Healthcare Network

3.1 Introduction to the Case Study

The primary motivation for this case study is to better understand the requirements for a practical framework for expressing and analyzing enterprise administrative security policies. This case study also serves as a thorough test case for testing our algorithm. The motivation for a case study based on healthcare information systems comes from the United States Health and Human Services Initiative for a Nationwide Health Information Network [The]. A healthcare network is an association of healthcare facilities with a common governing administration.

This case study involves of four organizations: Nationwide Health Information Network (NHIN), Healthcare Network, Patient Demographic Services (PDS), and Health and Human Services (HHS). PDS maintains demographic information about users, HHS is the main organization that sets standards and approves registration authorities, which provide signed credentials to health care providers. This case study focuses on policies for NHIN and Healthcare Network, because they are the organizations that maintain patient health care information.

NHIN is a central nationwide body that stores patient health records. Local health organizations interact with NHIN to obtain and report patient information, so that the NHIN has a summary record of every patient's healthcare history. Part of our policy for NHIN is transcribed from the policy for the Spine in [Bec05], which is based on the Output Based Specification Version 2.0 (OBS) [Nat03]. For completeness, we have written a

NHIN policy for this case study. However, we chose to omit it from the thesis because it is quite similar to the Spine policy i [Bec05], somewhat similar to although simpler than our HCN policy, and does not motivate or illustrate any additional features of our policy language or analysis algorithm. Individual healthcare facilities maintain detailed health records for their patients. We consider two representative facilities, a Substance Abuse Facility `getCleanSaf` and a Hospital `getWellHosp`.

As mentioned earlier, Healthcare Network and NHIN interact with each other to maintain an accurate record of patient healthcare history. To do so safely, they maintain a trust relationship that requires signed credentials from each side to prove the identity and attributes of the individual accessing the patient information. These credentials are issued by Registration Authorities (RAs) who are trusted by both parties. The details of the policy rules controlling these interactions are similar to policy rules for trust negotiation through registration authorities in [Bec05], so we omit these from the thesis as well. However, note that the interactions between NHIN and healthcare network motivated us to consider both organizations policies, because they both pertain to patient information access.

The rest of the chapter is organized as follows. Section 3.2 covers part of the Health Insurance and Portability and Accountability Act (HIPAA), which dictates the legal use and disclosure of patient health information in United States and is a motivating source for some of the policies in the case study. Section 3.3 explains how Role-Based Access Control (RBAC) is modeled in our framework, since part of the case study policy is role-based. The remaining sections contain details of the healthcare network policy.

3.2 Patient Confidentiality Under Health Insurance and Portability and Accountability Act (HIPAA)

The purpose of the United States Health and Human Services (HHS) Privacy Rule [Uni03] is to implement the requirements of HIPAA (1996). The Rule addresses the use and disclosure of individuals' health information, referred to as "protected health information", by organizations that are subject to the rule, referred to as "covered entities". One of the goals while framing the Rule was to strike a balance that permits important uses of the information, while protecting the privacy of people seeking health care.

Covered Entities. Health plans, health care clearinghouses, and any other health care provider who transmits health information in electronic form in connection with transactions for which the Secretary to HHS has adopted standards under HIPAA, is covered under the Rule. In our case study, `getWellHosp` is a covered entity falling under the Privacy Rule definition of health care provider, and the NHIN is a covered entity, falling under the definition of health care clearinghouse as a community health information management system. Note that Registration Authorities are not covered entities , because they do not process any individually identifiable information and do not fall under the definition of a business associate.

Protected Health Information (PHI). As per [Uni03], "all *indiviually identifiable information* held or transmitted by a covered entity or its business associate, in any form or media, whether electronic, paper, or oral is called *protected health information (PHI)*". Individually identifiable health information is patient information, including demographic data, that relates to:

- the individual's physical or mental health condition,
- the provision of health care to the individual, or
- payment for the provision of health care to the individual
- common identifiers such as name, address, birth date, Social Security Number.

[Uni03] also defines *De-Identified Health Information*, which can be used or disclosed without restrictions. It is health information that "neither identifies nor provides a reasonable basis to identify an individual". Information can be de-identified by the removal of the specified identifiers of the individual and of the individual's relatives, household members and employers. Further details on de-identifying patient data are given under end note 15 of [Uni03]. Policy to release de-identified information to designated users in Healthcare Network is covered under rules 3.5.32 and 3.5.33.

Permitted Uses and Disclosures. A covered entity may use or disclose protected health information for the following (but not limited to) purposes or situations:

1. To the individual who is the subject of the information. In our case study, this policy has been enforced through rule 3.5.21, which allows facility policy officers to grant access to patients to their record.

2. For treatment, payment and health care operations. Generally, an entity may disclose PHI to another entity for treatment operations if both covered entities have or had a relationship with the individual and the PHI pertains to the relationship. However, there are stricter rules for some kinds of PHI. For example, most uses and disclosures of psychotherapy notes for treatment and other health care operations require the patient's authorization, except when the covered entity is the one that originated the notes and is using them for treatment (in other words, the entity is currently treating the patient). We implement the second part of this policy in the NHIN policy, such as if a patient health record item, requested by a clinician to read, is a high confidentiality topic (such as, psychotherapy) then the clinician is allowed to read it only if he is both the author of the item and is currently treating the subject patient at the organization recorded in the item under this specialty.

3. With the individual's consent. Therefore, the patient has the right to authorize usage of his information. In this case study, we look at one specific instance of this policy where the patient can grant or revoke consent to treatment to/from appropriate clinicians making them members of the treating clinician role for that patient, which grants those clinicians access to some of the patient's information (see rules 3.5.5, 3.5.6, 3.5.7, 3.5.8, 3.5.11, 3.5.13).

4. As a special case, under the circumstances where the individual is incapacitated (in an emergency situation), the Rule allows the covered entity treating the individual to exercise its professional judgment in releasing only the patient's condition to people asking for it by patient name. Health care personnel treating a patient in emergency may also access patient's PHI to obtain suitable contact information and grant permission to view information about the patient's condition to the patient's relatives. This last part of the policy is implemented in rules 3.5.23 and 3.5.24 in our healthcare network policy.

This list is not exhaustive. For example, a covered entity may also disclose PHI under other circumstances, such as public interest and benefit activities. Such situations are not considered in this case study.

3.3 Representation of Role-Based Access Control

Role-based access control (RBAC) can be expressed in our framework. This section describes how some features of RBAC are modeled in this case study.

Role Membership. Role membership is represented by the intensional relation `memberOf(User, Role)`. The extensional predicate `directMemberOf(User, Role)` is the direct (i.e., not including inheritance) user-role assignment. Therefore, the initial policy includes the following rule which defines a user U as a `memberOf` a role R if U is a `directMemberOf` R. This allows users to be directly assigned to a role by adding facts to `directMemberOf`.

```
(3.3.1)
memberOf(User, Role) :- directMemberOf(User, Role)
```

Role Activation. A member of a role must activate the role to use the permissions granted to that role [SCFY96a]. Traditionally, a role is activated in the context of a particular session. Following [Bec05], we omit the concept of sessions.

Activation of role `Role` for user `User` is expressed by adding the fact `hasActivated(User, Role)` to the extensional relation `hasActivated`. Role deactivation is expressed by removing the corresponding `hasActivated` fact.

A user can activate a role he is a member of and deactivated any activated roles for himself.

```
(3.3.2)
permit(User, addFact(hasActivated(User, Role)))
:- memberOf(User, Role)
```

```
(3.3.3)
permit(User, removeFact(hasActivated(User, Role)))
:- hasActivated(User, Role)
```

3.4 Healthcare Network

The Healthcare Network is a representative case study of healthcare organizations which are typically composed of several facilities sharing information and operating under a larger organization-wide policy. We consider two specific facilities: `getCleanSaf`, and `getWellHosp`.

The Network Policy Administrator decides the administrative policy for all the facilities under the network. Each facility has its own Facility Policy Officer who sets the policy for the facility, consistent with the administrative policy set by the Network Policy Administrator.

3.4.1 Facilities

This section discusses the various facilities within the Network, and their functions.

`getCleanSaf`Facility

`getCleanSaf`facility treats patients substance abuse problems. This facility has only out-patients. Patients may undergo behavioral therapy and pharmacological treatment. There are multiple teams within the facility, each composed of psychiatrists, doctors and nurses. Each team may treat multiple patients.

The access policy for patient records at this facility is stricter than the policy at `getWellHosp`. Emphasis is on patient granting consent to treatment to a user in order for that user to get access to more information about the patient. Generally, only treating clinicians are granted privileges to perform operations that affect access to patient's records, such as creating team encounters.

`getWellHosp`

`getWellHosp` is a more diverse patient treatment facility. It has two kinds of workgroups: teams and wards. A team may consist of clinicians, nurses, and other employees of `getWellHosp`. `getWellHosp` also has multiple wards. Each in-patient is associated with a ward. The members of a ward are nurses, led by a head nurse. The administrative policy for `getWellHosp` is less strict than for `getCleanSaf`. For example, at `getWellHosp`, all clinicians can create patient encounters.

3.4.2 Patient Record

Records for patients of all facilities in the healthcare network are stored in a single database. We refer to patient records at the healthcare network level as local health records. A patient's local health record consists of a set of items of the form `(ID, Patient, Author, Topic, HealthData, Facility, EncounterID, Annotation)`, where :

- `ID` is a unique identifier for the item
- `Patient` is the id of the patient whose record item it is
- `Author` is the id of the user who created the item
- `Topic` is the topic of the item
- `HealthData` is the detalied health information, possibly including symptoms, test results, diagnosis, prognosis, and treatment
- `Facility` is the facility where the patient was seen when this item was created
- `EncounterID` is a unique identifier for the encounter, at `Facility`, under which this item was created
- `Annotation` is a comment that can be added after the item was added to the database to a patient record item. This field is typically used by patients, agents of patients, or treating clinicians to add comments to existing record items.

In a scalable implementation, these items are logically in the extensional database of the policy, constituting the relation `patientRecordItem` stored in a DBMS interfaced to the policy evaluation engine. Mature logic programming systems such as XSB[XSB], support external storage of relations in DBMS. Note that the `HealthData` field, which contains the largest amount of data, would never need to be loaded into the policy evaluation engine, because it is not used in the policy rules (it is mentioned but not used). The `EncounterID`, together with the associated `closedEncounters` relation, described in section 3.5.5, is used to identify record items associated with a currently running encounter. This is necessary to express policy rule 3.5.23 in section 3.5.6. Record items are stored under the relation `patientRecordItem(ID, Author, Topic, HealthData, Facility, EncounterID, Annotation)`. Items are added by adding facts to this relation.

3.5 Healthcare Network Policy

The Healthcare Network has a Network Policy Administrator (NPA) who writes the initial policy for all the facilities. Note that policy initialization is not itself controlled by any policy in the framework (this is a fundamental bootstrapping issue) and should be subject to other access controls and auditing (a consequence of this is that it is preferable to keep the initial policy small, and let the policy grow and evolve through administrative operations permitted by the policy.). The initial policy grants privileges to each Facility Policy Officer to modify the policy for his facility. This section presents the initial policy written by the NPA for each facility. Some of these rules apply to all facilities; some apply only to the specified facility.

3.5.1 Roles

There are network-wide roles that are common to all facilities within the network. Some are parameterized by a `Facility` argument, indicating the facility to which it applies. The `patient` role is the same for all facilities. A user is a member of the `patient` role if he is a registered patient for the network. The roles in the healthcare network policy are:

- `patient`
- `policyOfficer(Facility)` : role for policy officers at the specified facility
- `hrManager(Facility)` : role for Human Resource Managers at the specified facility. An HR Manager is responsible for creating workgroups and appointing workgroup heads.
- `clinician(Facility, Specialty)` : role for clinicians with the specified specialty at the specified facility
- `agent(Patient)` : role for agents of the specified patient
- `receptionist(Facility)` : role for receptionists at the specified facility
- `researcher(Facility)` : role for researchers at the specified facility
- `researchHead(Facility)` : the head of research at the specified facility
- `treatingClinician(Patient, Fac)` : role for clinicians treating the specified patient at the specified facility
- `workgroup(Wkgp, Facility, Specialty, WkgpType)` : a workgroup is a collection of facility employees, of varying specialties, that work

together on several patient encounters. Workgroup membership is represented by the membership in the `workgroup` role. A user is a member of `workgroup(Wkgp, Fac, Spcty, WkgpType)` if he is a member of the workgroup `Wkgp` at the facility `Fac` under specialty `Spcty`. The `WkgpType` argument is included to provide the facilities with the flexibility to further classify workgroups in to locally defined types. For example, `getWellHosp` has two types of workgroups, `team` and `ward`. A `team` may handle in-patients and out-patients, while a `ward` may handle only in-patients. The `policyOfficer(getWellHosp)` may set different policies for these two types of workgroups.

- `workgroupHead(Wkgp, Facility)` : role for the head of the workgroup `Wkgp` at facility `Fac`

The healthcare network allows patients to indicate their relatives. This helps identify suitable emergency contacts. This information is captured in the relation `relative(`$\texttt{Patient}_1$`, `$\texttt{Patient}_2$`)` implying that $\texttt{Patient}_1$ listed $\texttt{Patient}_2$ as a relative. Alternatively, this property could be expressed using a new role `relative(Patient)`. We chose to define it as a relation, because such a role would not need to be activated, and with such limited power it is simpler to express it as a relation. Also, note that `relative` is not necessarily a symmetric relation.

Statically Mutually Exclusive Roles (SMER) Constraints. Although we do not use SMER constraints in this case study, they can easily be expressed in ACAR. For example, if it is required that the `hrManager(Facility)` and `clinician(Facility, Spcty)` be statically mutually exclusive, then this can be achieved by enforcing this premise in the `addFact` permission rule for assigning a user as a direct member of these roles. This role assignment would be performed by a member of the `hrManager(Facility)` role:

```
permit(HR, addFact(directMemberOf(Cli, clinician(Facility, Spcty))))
:- memberOf(HR, hrManager(Facility)),
   ! directMemberOf(Cli, hrManager(Facility))

permit(User, addFact(directMemberOf(HR', hrManager(Facility))))
:- memberOf(User, hrManager(Facility)),
   ! directMemberOf(HR', clinician(Facility, _))
```

3.5.2 Workgroup Management Policy

As described in section 3.5.1, each facility has workgroups, which are collections of facility employees that work together on patient encounters. The network policy administrator specifies administrative policy for creation and management of workgroups, allowing facility policy officers to add rules for assigning users to workgroups and to appoint workgroup heads. This gives the facility policy officer flexibility to specialize the rules (e.g., with additional premises, representing requirements specific to the facility) when adding them.

A `policyOfficer` may add rules that grant the `hrManager` permission to appoint workgroup heads (3.5.1) and assign users to workgroups (3.5.2) for his facility, while including the premise that the `hrManager` himself is not a member of that workgroup. Such a permission is not granted directly in the administrative policy, by the network policy administrator, so that the `policyOfficer` can specialize it for specific workgroup types.

```
(3.5.1)
permit(User, addRule(
              permit(HRM,
                     addFact(directMemberOf(Head,
                                            workgroupHead(Wkgp, Fac))))
              :- memberOf(HRM, hrManager(Fac)),
                 ! directMemberOf(
                            HRM,
                            workgroup(Wkgp, Fac, Spcty, WkgpType)),
                 memberOf(Head,
                        workgroup(Wkgp, Fac, Spcty, WkgpType))))
:- hasActivated(User, policyOfficer(Fac))


(3.5.2)
permit(User, addRule(
              permit(HRM,
                     addFact(directMemberOf(
                                   Cli,
                                   workgroup(Wkgp, Fac, Spcty,
                                             WkgpType))))
              :- memberOf(HRM, hrManager(Fac))))
:- hasActivated(User, policyOfficer(Fac))
```

A `policyOfficer` may appoint an HR manager for his facility. This is done by assigning the role `hrManager(Fac)` to the user.

```
(3.5.3)
permit(User, addFact(directMemberOf(HRM, hrManager(Fac)))
:- hasActivated(User, policyOfficer(Fac))
```

A `policyOfficer` may add rules that allow a workgroup head to assign users to his workgroup.

```
(3.5.4)
permit(User, addRule(
              permit(Head,
                     addFact(
                      directMemberOf(User', workgroup(Wkgp,
                                                      Fac, Spcty,
                                                      WkgpType))))
              :- memberOf(Head, workgroupHead(Wkgp, Fac)))
:- hasActivated(User, policyOfficer(Fac))
```

3.5.3 Consent To Treatment Policy

Patients, and certain designated users, may grant clinicians consent to treat the patient. This is represented by the relation `consentToTreatment(Pat, Cli, Fac)`, which means that clinician `Cli` has consent to treat patient `Pat` at facility `Fac`.

`getCleanSaf` Consent To Treatment Policy

At `getCleanSaf`, consent to treatment can be granted only by the patient himself. `policyOfficer(getCleanSaf)` can add rules allowing patients to add `consentToTreatment (Pat, Cli, getCleanSaf)` facts. Note that the facility argument has been instantiated with `getCleanSaf`, which ensures that the patient does not accidentally grant consent outside of the facility.

```
(3.5.5)
permit(User,
       addRule(
         permit(Pat,
                addFact(consentToTreatment(Pat, Cli,
                                           getCleanSaf)))
         :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getCleanSaf))
```

Correspondingly, `policyOfficer(getCleanSaf)` may add rules allowing patients to revoke consent to treatment.

```
(3.5.6)
permit(User,
       addRule(
         permit(Pat,
                removeFact(consentToTreatment(Pat, Cli,
                                               getCleanSaf)))
         :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getCleanSaf))
```

getWellHosp **Consent To Treatment Policy**

At getWellHosp, consent the treatment may be granted in a variety of ways. The patient may grant consent to treatment, as at getCleanSaf. In addition, an agent of the patient can grant the consent.

policyOfficer(getWellHosp) can add rules that allow patients to grant and revoke consent to treatment to clinicians at getWellHosp.

```
(3.5.7)
permit(User,
       addRule(
        permit(Pat,
               addFact(consentToTreatment(Pat, Cli, getWellHosp)))
        :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

```
(3.5.8)
permit(User,
       addRule(
        permit(Pat,
               removeFact(consentToTreatment(Pat, Cli, getWellHosp)))
        :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

policyOfficer(getWellHosp) can add rules that allow patient's agents to grant and revoke consent to treatment to clinicians at getWellHosp.

```
(3.5.9)
permit(User,
       addRule(
         permit(Ag,
                addFact(consentToTreatment(Pat, Cli, getWellHosp)))
         :- hasActivated(Ag, agent(Pat))))
:- hasActivated(User, policyOfficer(getWellHosp))
```

```
(3.5.10)
permit(User,
       addRule(
        permit(Ag,
               removeFact(consentToTreatment(Pat, Cli, getWellHosp)))
        :- hasActivated(Ag, agent(Pat))))
:- hasActivated(User, policyOfficer(getWellHosp))
```

getWellHosp's policy for registering agents appears in Section 3.5.7.

3.5.4 Treating Clinician Policy

The fact that a clinician is treating a patient is represented by making that clinician a member of an appropriate instance of the treatingClinician role. Specifically, memberOf(Cli, treatingClinician(Pat, Fac, Spcty)) means that the clinician Cli is a treating clinician for patient Pat at facility Fac under specialty Spcty. This information is used in inferring access control permissions to patient record, as detailed in Section 3.5.6. Different facilities have different definitions for this relation, as described below. Thus, there are different administrative policy rules for how a facility policy officer may define membership in the treatingClinician role.[1]

getCleanSaf Treating Clinician Policy

At getCleanSaf, a clinician may be a treatingClinician for a patient if he has been given consent to treatment for that patient. The facility policy officer may, therefore, add policy rules defining this relation provided they include the consentToTreatment premise.

```
(3.5.11)
permit(User,
       addRule(
         memberOf(Cli, treatingClinician(Pat, getCleanSaf))
         :- consentToTreatment(Pat, Cli, getCleanSaf)))
:- hasActivated(User, policyOfficer(getCleanSaf))
```

getWellHosp Treating Clinician Policy

At getWellHosp, a clinician may be considered a treating clinician for a patient if he has activated the clinician role and is a member of a workgroup

[1]The treating clinician policy presented in this section is the same as presented earlier in the example in Section 2.1

which is assigned an encounter for that patient. The encounter creation policy for `getWellHosp` is presented in section 3.5.15. A patient encounter is represented by the relation `encounter(EncID, Pat, Wkgp, Fac, Type)`, which means that there is an encounter identified by `EncID` of treatment type `Type` for patient `Pat` being handled by the workgroup `Wkgp` at facility `Fac`.

`policyOfficer(getWellHosp)` may add rules defining membership in the role `treatingClinician(Pat, getWellHosp)` provided they include the two premises described above.

```
(3.5.12)
permit(User,
       addRule(
         memberOf(Cli, treatingClinician(Pat, getWellHosp))
         :- hasActivated(Cli, clinician(getWellHosp, Spcty)),
            memberOf(Cli workgroup(Wkgp, getWellHosp, Spcty, WkgpType)),
            encounter(EncID, Pat, Wkgp, getWellHosp, Type)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

Also, `policyOfficer(getWellHosp)` may add rules that allow clinician who has explicit consent to treat the patient at `getWellHosp`to be a treating clinician for the patient.

```
(3.5.13)
permit(User,
       addRule(
         memberOf(Cli, treatingClinician(Pat, getWellHosp))
         :- consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

3.5.5 Encounter Administrative Policy

When a patient arrives at a facility, a workgroup is assigned to treat the patient by creating an encounter involving the patient and the workgroup. An encounter is created by adding a fact `encounter(EncID, Pat, Wkgp, Facility, Type)`, thereby assigning `Wkgp` at `Facility` to treat patient `Pat` under treatment type `Type`. Treatment type indicates the type of treatment procedure being conducted under the encounter. Examples of treatment types are `mri`, `xray`, `surgery`, and so on. `EncID` is a unique identifier that ids an encounter at a facility. The relation `closedEncounters(EncID)` records the closed encounters. Both `encounter` and `closedEncounters` are add-only extensional relations. This is in accordance with the design of most medical record systems, where information is added and never deleted.

The rest of this section covers the administrative policy for creating encounters at each facility and closing completed encounters.

`getCleanSaf` Encounter Administrative Policy

At `getCleanSaf`, only treating clinicians are permitted to create encounters for the patients they are treating. Thus, the `policyOfficer` for this facility is required to include `treatingClinician` role membership as a premise in the `addRule` permission rules for adding `encounter` facts. Since membership in `treatingClinician` role for `getCleanSaf`is inferred from `consentToTreatment` (as per rule 3.5.11), this gives the patient more control over who can create encounters for them at the facility.

```
(3.5.14)
permit(User,
       addRule(
         permit(Cli,
                addFact(encounter(EncID, Pat, Wkgp,
                                  getCleanSaf, Type)))
         :- memberOf(Cli,
                     treatingClinician(Pat, getCleanSaf))))
:- hasActivated(User, policyOfficer(getCleanSaf))
```

`policyOfficer(getCleanSaf)` may allow clinicians at `getCleanSaf` to close an encounter. An encounter is closed by adding a fact to the `closedEncounters` relation.

```
(3.5.15)
permit(User,
       addRule(
         permit(Cli,
                addFact(closedEncounters(EncID)))
         :- hasActivated(Cli, clinician(getCleanSaf, Spcty)),
            encounter(EncID, Pat, Wkgp, getCleanSaf, Type)))
:- hasActivated(User, policyOfficer(getCleanSaf))
```

`getWellHosp` Encounter Administrative Policy

At `getWellHosp`, all clinicians are allowed to create encounters for patients being treated at the facility. The reason for allowing only clinicians, not other staff (for example, receptionists), to create encounters is to ensure that an appropriate type of encounter, with an appropriate workgroup, is created, based on the patient's symptoms. Broadly speaking, creating an encounter gives the workgroup handling the encounter, access to some of the items in the patient's record.

The `policyOfficer(getWellHosp)` can add rules allowing addition of `encounter` facts, provided the rules include the premise that the user adding the fact has activated the `clinician(getWellHosp)` role.

```
(3.5.16)
permit(User,
       addRule(
         permit(Cli,
                addFact(encounter(EncID, Pat, Wkgp, getWellHosp, Type)))
         :- hasActivated(Cli, clinician(getWellHosp, Spcty)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

`policyOfficer(getWellHosp)` may allow any clinician at `getWellHosp` to close an encounter.

```
(3.5.17)
permit(User,
       addRule(
         permit(Cli, addFact(closedEncounters(EncID)))
         :- hasActivated(Cli, clinician(getWellHosp, Spcty)),
            encounter(EncID, Pat, Wkgp, getWellHosp, Type)))
:- hasActivated(User, policyOfficer(getWellHosp))
```

3.5.6 Patient Record Access Policy

The network's administrative policy that allows facility policy officer to add rules that grant access to patient records is the same for all facilities, so we use a variable `Fac` to represent the facility in the rules below.

Record items are stored under the relation `patientRecordItem(ID, Author, Topic, HealthData, Facility, EncounterID, Annotation)`. Items are added by adding facts to this relation. Items are added when a clinician is treating a patient under an encounter.

`policyOfficer` can allow clinicians treating patients at their facility to add new items for the treated patients if the associated encounter is not closed.

```
(3.5.18)
permit(User,
       addRule(
         permit(Cli,
                addFact(patientRecordItem(ID, Pat, Cli, Spcty,
                                            HealthData, Fac,
                                        EncID, Anno)))
         :- treatingClinician(Cli, Pat, Fact),
            hasActivated(Cli, clinician(Fac, Spcty)),
            !(closedEncounters(EncID)),
            encounter(EncID, Pat, Wkgp, Fac),
            memberOf(Cli, workgroup(Wkgp, Fac, Spcty, WkgpType))))
:- hasActivated(User, policyOfficer(Fac))
```

A patient record item is accessed by invoking the function `getRecordItemById(Id)`, where `Id` is the identifier for the item. The `policyOfficer` can allow a clinician to access a record item if one of the following conditions hold:

- the clinician is a treating clinician for the patient and has activated the `clinician` role with the specialty which matches the topic of the item (so that they have access only to information that is relevant to their expertise):

```
(3.5.19)
permit(User,
       addRule(
         permit(Cli, getRecordItemById(Id))
         :- hasActivated(Cli, clinician(Fac, Spcty)),
            memberOf(Cli, treatingClinician(Pat, Fac)),
            patientRecordItem(Id, Pat, Author, Spcty,
                              HealthData, Fac,
                              EncID, Annotation)))
:- hasActivated(User, policyOfficer(Fac))
```

- the clinician has activated the `clinician` role and is the author of the item. This policy allows clinicians to access items they authored, even if they are no longer treating the patient:

```
(3.5.20)
permit(User,
       addRule(
         permit(Cli, getRecordItemById(Id))
         :- hasActivated(Cli, clinician(Fac, Spcty)),
            patientRecordItem(Id, Pat, Cli, Spcty,
                              HealthData, Fac,
                              EncID, Annotation)))
:- hasActivated(User, policyOfficer(Fac))
```

`policyOfficer` may add rules allowing patients (3.5.21) and their agents (3.5.22) to access the patient's record items.

```
(3.5.21)
permit(User,
       addRule(
         permit(Pat, getRecordItemById(Id))
         :- hasActivated(Pat, patient),
            patientRecordItem(Id, Pat, Cli, Spcty,
                              HealthData, Fac,
                              EncID, Annotation)))
:- hasActivated(User, policyOfficer(Fac))
```

```
(3.5.22)
permit(User,
       addRule(
         permit(Ag, getRecordItemById(Id))
         :- hasActivated(Ag, agent(Pat)),
            patientRecordItem(Id, Pat, Cli, Spcty,
                              HealthData, Fac,
                              EncID, Annotation)))
:- hasActivated(User, policyOfficer(Fac))
```

The Privacy Rule [Uni03] specifies that in case of an emergency, the covered entity treating the patient may release patient health information to relatives listed by the patient in the facility directory. Under our interpretation of this rule, this is achieved by adding the fact `releaseToRelatives(Id)`, which enables release of the health data under the item with identifier `Id`, but only if `Id` is an item created under a current encounter. The relatives can access the health data by invoking `getHealthData(Id)` method. The `policyOfficer` may add rules allowing users to add `releaseToRelatives` facts (3.5.23) and rules granting access to relatives for released items (3.5.24)..

```
(3.5.23)
permit(User,
       addRule(
         permit(U, addFact(releaseToRelatives(Id)))
         :- patientRecordItem(Id, Pat, Cli, emergency,
                              HealthData, Fac,
                              EncID, Annotation)))
:- hasActivated(User, policyOfficer(Fac))
```

```
(3.5.24)
permit(User,
       addRule(
         permit(Rel, getHealthData(Id))
         :- patientRecordItem(Id, Pat, Cli, emergency,
                                HealthData, Fac,
                                EncID, Annotation),
            relative(Pat, Rel),
            releaseToRelatives(Id)))
:- hasActivated(User, policyOfficer(Fac))
```

3.5.7 Registration and Appointment of Users

This section presents the policy for patient and agent registrations and clinician appointments. The rules apply to all facilities.

Patient Registration: Patients are registered at a healthcare network by a `receptionist` at a facility. Note that there is only one unique registration for the patient for the entire network. A precondition for patient registration is that the patient should be a member of `patient` at NHIN.

```
(3.5.25)
permit(Rec, addFact(directMemberOf(Pat, patient)))
:- hasActivated(Rec, receptionist),
   nhin issues memberOf(Pat, patient)
```

Agent Registration: Patients can appoint and remove agents for themselves. In [Bec05] Caldicott Guardians can also register agents at local healthcare facilities. For brevity, we omit Caldicott Guardians from our case study.

```
(3.5.26)
permit(Pat, addFact(directMemberOf(Ag, agent(Pat))))
:- memberOf(Pat, patient)
```

```
(3.5.27)
permit(Pat, removeFact(directMemberOf(Ag, agent(Pat))))
:- memberOf(Pat, patient)
```

A user who is an agent for a patient at NHIN is automatically a member of `agent` for that patient at the healthcare network.

```
(3.5.28)
memberOf(Ag, agent(Pat))
:-  nhin issues memberOf(Ag, agent(Pat))
```

Clinician and Other Employee Registration: All employees at a facility are appointed by `hrManager`. We present a few specific cases here: `clinician` (3.5.29), `receptionist` (3.5.30) and `researchHead` (3.5.31). The network policy administrator allows the facility policy officer to add rules allowing members of `hrManager` role to appoint employees, because there might be additional restrictions local to the facility that the facility policy officer should impose. For example, for appointing clinicians, the network-wide policy allows clinicians that are federally certified to be appointed. However, the facility policy officer might need to add state-level certification requirements that are defined under the facility's policy and not under the network policy. These rules were used in chapter 2 to illustrate the policy language.

```
(3.5.29)
permit(User,
       addRule(
         permit(HR,
                addFact(directMemberOf(Cli,
                                       clinician(Facility, Spcty))))
         :- memberOf(HR, hrManager(Facility)),
            federalCertifiedClinician(Cli)))
:- hasActivated(User, policyOfficer(Facility))


(3.5.30)
permit(User,
       addRule(
         permit(HR,
                addFact(directMemberOf(Cli,
                                       receptionist(Facility))))
         :- memberOf(HR, hrManager(Facility))))
:- hasActivated(User, policyOfficer(Facility))


(3.5.31)
permit(User,
       addRule(
         permit(HR,
                addFact(directMemberOf(Cli,
                                       researchHead(Facility))))
         :- memberOf(HR, hrManager(Facility))))
:- hasActivated(User, policyOfficer(Facility))
```

3.5.8 Access to De-identified Patient Information

Researchers can read de-identified information (information that does not identify the individual it belongs to) through the `readDeidentified(Query)` operation. It takes as argument a query to select aggregate information on the record items. The current version of our policy does not explore the details of this argument, but a more detailed policy might allow different queries for different researchers. Permission to invoke this function, at each facility, is granted by `policyOfficer` to users belonging to `researcher` role:

```
(3.5.32)
permit(User, addRule(permit(Res, readDeidentified(Query))
                     :- hasActivated(Res, researcher(Fac))))
:- hasActivated(User, policyOfficer(Fac))
```

A `researchHead` for a facility can assign users to the `researcher` role:

```
(3.5.33)
permit(RH, addFact(directMemberOf(Res, researcher(Fac))))
:- hasActivated(RH, researchHead(Fac))
```

3.6 `getCleanSaf` Policy

`getCleanSaf`policy consists of the initial policy written by the network policy adminstrator and presented in section 3.5, plus rules added by `policyOfficer(getCleanSaf)` and presented in detail in this section, after the following overview.

The added policy rules for `consentToTreatment`, `treatingClinician` and encounter creation are the same as the patterns in the administrative policy rules in section 3.5 with `Facility` instantiated with `getCleanSaf`, since no additional premises are required. However, for access to patient record items, only clinicians who are treating clinicians under the same specialty as the topic of the items may access them. `policyOfficer( getCleanSaf)` adds this rule under administrative rule 3.5.19. `policyOfficer(getCleanSaf)` does not add any rules using administrative rule 3.5.20, to ensure that if the clinician is not a current treating clinician, then he cannot access patient record items, regardless of whether he authored those items.

Workgroup Management: `getCleanSaf`has teams as the sole type of workgroups. These teams are managed by the `hrManager(getCleanSaf)` consistent with the policy written by `policyOfficer(getCleanSaf)`.

The `hrManager(getCleanSaf)` may appoint a workgroup head from clinicians who are members of the workgroup under the specialty psychiatry.

```
(3.6.1 ; added by policyOfficer(getCleanSaf) via 3.5.1)
permit(HR,
       addFact(directMemberOf(Head,
                              workgroupHead(Wkgp,
                                            getCleanSaf))))
:- memberOf(HR, hrManager(getCleanSaf)),
   memberOf(Head, workgroup(Wkgp, getCleanSaf,
                            psychiatry, team))
```

The `hrManager(getCleanSaf)` may assign users to be members of teams. Clinicians with specialty `pharmacology` (3.6.2), `nursing` (3.6.3), or `psychiatry` (3.6.4) are allowed on the teams.

```
(3.6.2 ; added by policyOfficer(getCleanSaf) via 3.5.2)
permit(HR,
       addFact(directMemberOf(Cli,
                              workgroup(Wkgp,
                                        getCleanSaf,
                                        pharmacology, team))))
:- memberOf(HR, hrManager(getCleanSaf))
```

```
(3.6.3 ; added by policyOfficer(getCleanSaf) via 3.5.2)
permit(HR,
       addFact(directMemberOf(Cli,
                              workgroup(Wkgp,
                                        getCleanSaf,
                                        nursing, team))))
:- memberOf(HR, hrManager(getCleanSaf))
```

```
(3.6.4 ; added by policyOfficer(getCleanSaf) via 3.5.2)
permit(HR,
       addFact(directMemberOf(Cli,
                              workgroup(Wkgp,
                                        getCleanSaf,
                                        psychiatry, team))))
:- memberOf(HR, hrManager(getCleanSaf))
```

A workgroup head may also assign clinicians as members of his team, if the clinicians have one of the specialties: `pharmacology` (3.6.5), `nursing` (3.6.6), or `psychiatry` (3.6.7).

```
(3.6.5 ; added by policyOfficer(getCleanSaf) via 3.5.4)
permit(Head,
       addFact(directMemberOf(Cli,
                              workgroup(Wkgp,
                                        getCleanSaf,
                                        pharmacology, team))))
:- memberOf(Head, workgroupHead(Wkgp, getCleanSaf))


(3.6.6 ; added by policyOfficer(getCleanSaf) via 3.5.4)
permit(Head,
       addFact(directMemberOf(Cli,
                              workgroup(Wkgp,
                                        getCleanSaf,
                                        nursing, team))))
:- memberOf(Head, workgroupHead(Wkgp, getCleanSaf))


(3.6.7 ; added by policyOfficer(getCleanSaf) via 3.5.4)
permit(Head,
       addFact(directMemberOf(Cli,
                              workgroup(Wkgp,
                                        getCleanSaf,
                                        psychiatry, team))))
:- memberOf(Head, workgroupHead(Wkgp, getCleanSaf))
```

Consent To Treatment: A patient may grant consent to treatment to a clinician for `getCleanSaf`.

```
(3.6.8 ; added by policyOfficer(getCleanSaf) via 3.5.5)
permit(Pat,
       addFact(consentToTreatment(Pat, Cli,
                                  getCleanSaf)))
:- hasActivated(Pat, patient)
```

Treating Clinician: A clinician is a treating clinician for a patient at `getCleanSaf`if he has been granted consent to treatment for that patient.

```
(3.6.9 ; added by policyOfficer(getCleanSaf) via 3.5.11)
memberOf(Cli, treatingClinician(Pat, getCleanSaf))
:- consentToTreatment(Pat, Cli, getCleanSaf)
```

Encounter Creation: A clinician may create an encounter for a patient at `getCleanSaf`if he is a treating clinician for that patient and there is no existing encounter with the same encounter ID.

```
(3.6.10 ; added by policyOfficer(getCleanSaf) via 3.5.14)
permit(Cli,
       addFact(encounter(EncID, Pat, Wkgp,
                         getCleanSaf, Type)))
:- memberOf(Cli, treatingClinician(Pat, getCleanSaf)),
   ! encounter(EncID, _, _, _, _)
```

Closing Encounters: Workgroup heads may close encounters involving workgroup that they head.

```
(3.6.11 ; added by policyOfficer(getCleanSaf) via 3.5.15)
permit(Head, addFact(closedEncounters(EncID)))
:- hasActivated(Head, clinician(getCleanSaf, Spcty)),
   encounter(EncID, Pat, Wkgp, getCleanSaf, Type),
   memberOf(Head, workgroupHead(Wkgp, getCleanSaf))
```

Patient Record Access: Only the clinicians who are treating the patient may obtain record items, whose topic matches their active specialty, for that patient.

```
(3.6.12 ; added by policyOfficer(getCleanSaf) via 3.5.19)
permit(Cli, getRecordItemById(Id))
:- hasActivated(Cli, clinician(Fac, Spcty)),
   memberOf(Cli, treatingClinician(Pat, Fac)),
   patientRecordItem(Id, Pat, Author, Spcty,
                         HealthData, Fac, EncID, Annotation)
```

Only clinicians treating the patient under the specialty `emergency` may release emergency healthdata items to relatives. This is done by adding `releaseToRelatives` facts. Further, to ensure that information about past emergency encounters is not released, `policyOfficer(getCleanSaf)` adds an additional premise that the associated encounter is not closed.

```
(3.6.13 ; added by policyOfficer(getCleanSaf) via 3.5.23)
permit(Cli, addFact(releaseToRelatives(Id)))
:- patientRecordItem(Id, Pat, Cli, emergency, HealthData,
                         getCleanSaf, EncID,
                         Annotation),
   memberOf(Cli, treatingClinician(Pat, getCleanSaf)),
   !(closedEncounters(EncID)),
   encounter(EncID, Pat, Wkgp, getCleanSaf, emergency),
   memberOf(Cli, workgroup(Wkgp, getCleanSaf,
            emergency, WkgpType))
```

`policyOfficer` also adds a rule allowing relatives to invoke the `getHealthData` operation on a patient's record items which have been released.

```
(3.6.14 ; added by policyOfficer(getCleanSaf) via 3.5.24)
permit(Rel, getHealthData(Id))
:- patientRecordItem(Id, Pat, Cli, ‘A-and-E’, HealthData,
                     getCleanSaf, EncID,
                     Annotation),
   relative(Pat, Rel),
   releaseToRelatives(Id)
```

3.7 `getWellHosp` Policy

`getWellHosp`policy consists of the initial policy written by the network policy administrator and presented in section 3.5, plus rules added by `policyOfficer(getWellHosp)` and, presented in detail in this section after the following overview.

The added policy rules for `consentToTreatment`, `treatingClinician` and patient record access are the same as the patterns in the administrative policy rules in section 3.5 with `Facility` instantiated with `getWellHosp`. For encounter creation, `policyOfficer(getWellHosp)` adds two rules which distinguish between encounters created for workgroups of type `team` and another rule for creation of encounters for workgroups of type `ward`. Depending on the type of the workgroup, `policyOfficer(getWellHosp)` imposes additional premises on who can create an encounter.

Workgroup Management: As mentioned in section 3.5.2, `getWellHosp` has two types of workgroups: wards and teams. While a team consists of clinicians of varying specialties, a ward consists of clinicians working under the specialty `nursing` in that ward. `hrManager(getWellHosp)` appoints heads for both teams and wards, and may appoint members for these workgroups, as well. Additionally, team heads may appoint members for their team. Ward heads are not allowed to appoint members for their ward, because wards are typically set up by `getWellHosp`'s human resource administration, represented in our policy by the HR manager.

A member of the role `hrManager(getWellHosp)` may appoint heads of workgroups from amongst the clinicians working at `getWellHosp`.

```
(3.7.1 ; added by policyOfficer(getWellHosp) via 3.5.1)
permit(HR,
       addFact(directMemberOf(Head, workgroupHead(Wkgp, getWellHosp))))
:- memberOf(HR, hrManager(getWellHosp)),
   memberOf(Head, clinician(getWellHosp, Spcty)),
   memberOf(Head, workgroup(Wkgp, getWellHosp, Spcty, WkgpType))
```

The `hrManager(getWellHosp)` may assign `getWellHosp` clinicians to workgroups, under a specialty that the clinician possesses. The workgroup can be a team or ward.

```
(3.7.2 ; added by policyOfficer(getWellHosp) via 3.5.2)
permit(HR,
       addFact(directMemberOf(Cli, workgroup(Wkgp, getWellHosp, Spcty,
                                              WkgpType))))
:- memberOf(HR, hrManager(getWellHosp)),
   memberOf(Cli, clinician(getWellHosp, Spcty))
```

For workgroups of type `team`, the workgroup head may assign `getWellHosp` clinicians to the team he is heading, under a specialty that the clinician possesses.

```
(3.7.3 ; added by policyOfficer(getWellHosp) via 3.5.4)
permit(Head,
       addFact(directMemberOf(Cli, workgroup(Wkgp, getWellHosp, Spcty,
                                             team))))
:- workgroupHead(Head, Wkgp, team),
   memberOf(Cli, clinician(getWellHosp, Spcty))
```

Consent To Treatment: A patient can grant consent to treatment for himself to a clinician at `getWellHosp`.

```
(3.7.4 ; added by policyOfficer(getWellHosp) via 3.5.7)
permit(Pat, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Pat, patient)
```

An agent for a patient can also grant consent to treatment for that patient to a clinician at `getWellHosp`.

```
(3.7.5 ; added by policyOfficer(getWellHosp) via 3.5.9)
permit(Ag, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Ag, agent(Pat))
```

Treating Clinician: A clinician is a treating clinician at `getWellHosp`for a patient if his `clinician` role is active with a specialty under which he is a member of a workgroup (at `getWellHosp`) which is handling an encounter for that patient.

```
(3.7.6 ; added by policyOfficer(getWellHosp) via 3.5.12)
memberOf(Cli, treatingClinician(Pat, getWellHosp))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   memberOf(Cli, workgroup(Wkgp, getWellHosp, Spcty, WkgpType)),
   encounter(EncID, Pat, Wkgp, getWellHosp, Type),
   ! closedEncounters(EncID)
```

Encounter Creation: For workgroups of type `team`, any team member may create a `getWellHosp`encounter for a patient if they have the consent to treat that patient, and associate it with their team, if there is no existing encounter with the same encounter ID.

```
(3.7.7 ; added by policyOfficer(getWellHosp) via 3.5.16)
permit(Cli, addFact(encounter(EncID, Pat, Team, getWellHosp, Type)))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   memberOf(Cli, workgroup(Team, getWellHosp, Spcty, team))
   consentToTreatment(Pat, Cli, getWellHosp),
   ! encounter(EncID, _, _, _, _)
```

For wards, only the head of a ward may create encounters involving the ward.

```
(3.7.8 ; added by policyOfficer(getWellHosp) via 3.5.16)
permit(Cli, addFact(encounter(EncID, Pat, Ward, getWellHosp, Type)))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   memberOf(Cli, workgroupHead(Ward, getWellHosp)),
   memberOf(Cli, workgroup(Ward, getWellHosp, Spcty, ward)),
   ! encounter(EncID, _, _, _, _)
```

Closing Encounters: Workgroup heads may close encounters involving the workgroup that they head.

```
(3.7.9 ; added by policyOfficer(getWellHosp) via 3.5.17)
permit(Head, addFact(closedEncounters(EncID)))
:- hasActivated(Head, clinician(getWellHosp, Spcty)),
   encounter(EncID, Pat, Wkgp, getWellHosp, Type),
   memberOf(Head, workgroupHead(Wkgp, getWellHosp))
```

Patient Record Access: A treating clinician for a patient may obtain a record item of that patient if the clinician has activated the `clinician` role with the specialty that matches the topic of the item.

```
(3.7.10 ; added by policyOfficer(getWellHosp) via 3.5.19)
permit(Cli, getRecordItemById(Id))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   memberOf(Cli, treatingClinician(Pat, getWellHosp)),
   patientRecordItem(Id, Pat, Author, Spcty, HealthData, getWellHosp,
                     EncID, Annotation)
```

A clinician may obtain a patient's `getWellHosp`record item if he is the author of that item and has activated the `clinician` role with the specialty matching the topic of the item.

```
(3.7.11 ; added by policyOfficer(getWellHosp) via 3.5.20)
permit(Cli, getRecordItemById(Id))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   patientRecordItem(Id, Pat, Cli, Spcty, HealthData, getWellHosp,
                     EncID, Annotation)
```

`getWellHosp`policy for releasing information about emergency encounters to relatives is the same as the `getCleanSaf`'s policy on the matter.

```
(3.7.12 ; added by policyOfficer(getWellHosp) via 3.5.23)
permit(Cli, addFact(releaseToRelatives(Id)))
:- patientRecordItem(Id, Pat, Cli, ‘A-and-E’, HealthData, getWellHosp,
                     EncID, Annotation),
   memberOf(Cli, treatingClinician(Pat, getWellHosp)),
   !(closedEncounters(EncID)),
   encounter(EncID, Pat, Wkgp, getWellHosp, emergency),
   memberOf(Cli, workgroup(Wkgp, getWellHosp, ‘A-and-E’, WkgpType))
```

```
(3.7.13 ; added by policyOfficer(getWellHosp) via 3.5.24)
permit(Rel, getHealthData(Id))
:- patientRecordItem(Id, Pat, Cli, ‘A-and-E’, HealthData, getWellHosp,
                     EncID, Annotation),
   relative(Pat, Rel),
   releaseToRelatives(Id)
```

Chapter 4

Abductive Analysis of Administrative Policies in Rule-based Access Control

The policy language ACAR described in chapter 2 allows administrative policies to be expressed concisely and at a desirable level of abstraction compared to ARBAC. However, fully understanding the implications of a rule-based administrative policy in ACAR is even more difficult 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 consider possible chains of inferences using rules in each intermediate policy. This chapter presents a symbolic analysis algorithm for answering abductive atom-reachability queries for ACAR policies, i.e. for determining whether changes by a specified administrators can lead to a policy in which some instance of a specified atom, called the goal, is derivable. As discussed in chapter 1, abductive analysis means that the algorithm does not require complete information about facts in the initial policy; rather, the algorithm computes minimal set of facts that, if present in the initial policy, imply reachability of the goal.

4.1 Background on Abduction

Philosopher Charles Peirce first introduced the notion of abduction. [KKT92] defines abduction as the "probational adoption of a hypothesis" as explanation for observed facts (results), according to known laws. Abduction is widely used in common-sense reasoning, for instance in diagnosis, to reason from effect to cause [CM86, CW89]. Following is an example of abductive reasoning presented in [KKT92] and drawn from [Pea87]. Consider the following theory T

$$\begin{aligned} grassIsWet &\leftarrow rainedLastNight \\ grassIsWet &\leftarrow sprinklerWasOn \\ shoesAreWet &\leftarrow grassIsWet. \end{aligned}$$

If we observe that shoes are wet, and we want to know why this is so, $rainedLastNight$ is a possible explanation, that is, a set of hypotheses, that together with the known laws in T implies the given observation. $sprinklerWasOn$ is another explanation.

Abduction in logic is defined as follows in [KKT92].

Given a set of rules and facts T (a theory presentation), and a fact G (observation), to a first approximation, the abductive task can be characterized as the problem of finding a set of facts Δ (abductive explanation for G) such that:

(1) $T \cup \Delta \vdash G$,

(2) $T \cup \Delta$ is consistent.

This characterization of abduction is independent of the language in which T, G and Δ are formulated. A set of rules and facts S is consistent if for any fact f, $S \vdash f \Rightarrow S \not\vdash !f$. Note that with this unrestricted definition of Δ we may get results which simply explain one effect in terms of another effect. If we want to convey the result in terms of the "cause" of the effect, some restrictions need to be imposed on Δ. Therefore, explanations are often restricted to belong to a special pre-specified domain-specific class of sentences called *abducibles*. We represent the abducibles as A. A set of ground facts Δ is abducible if it conforms to the set of facts represented by A. We explain the specification of abducible atoms A in more detail in section 4.2.

4.2 Abductive Reachability

This section defines abductive atom-reachability queries and their solutions.

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 $a = b\theta$. Recall from section 2.2 that the policy language, and hence the set of possible substitutions θ, is parameterized by the set of constructors, and that in examples, we take that set of constructors to be the set of constructors that appear in the given problem instance. For an atom a and a set A of atoms, let $[\![a]\!] = \{a' \mid a' \preceq a\}$ and $[\![A]\!] = \bigcup_{a \in A} [\![a]\!]$.

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 $a \in [\![\langle Ab, nAb \rangle]\!]$, where $[\![\langle Ab, nAb \rangle]\!] = [\![Ab]\!] \setminus [\![nAb]\!]$.

A *goal* is an atom.

Given an initial policy P_0 and a set U_0 of users (the active administrators), the *state graph* for P_0 and U_0, denoted $\mathrm{SG}(P_0, U_0)$, contains policies reachable from P_0 by actions of users in U_0. Specifically, $\mathrm{SG}(P_0, U_0)$ is the least graph (N, E) such that (1) $P_0 \in N$ and (2) $\langle P, U : op, P' \rangle \in E$ and $P' \in N$ if $P \in N \wedge U \in U_0 \wedge \langle P, U : op, P' \rangle \in T$.

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 a goal. Informally, P_0 contains rules and facts that are definitely present in the initial state, and $[\![A]\!]$ 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 $\langle \Delta, G \rangle$ such that Δ is a ground subset of $[\![A]\!]$, G is a ground instance of G_0, and $\mathrm{SG}(P_0 \cup \Delta, U_0)$ contains a policy P such that $P \vdash G$. Informally, a ground solution $\langle \Delta, G \rangle$ indicates that a policy P in which G holds is reachable from $P_0 \cup \Delta$ through administrative actions of users in U_0.

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

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

A *comprehensive solution* to an abductive atom-reachability query $\langle P_0, U_0, A, G_0 \rangle$ is a set S of tuples of the form $\langle \Delta, G, D \rangle$, where Δ is a set of atoms (not necessarily ground), G is an atom (not necessarily ground), and D is a set (interpreted as a conjunction) of tuple disequalities over the variables in Δ and G, such that (1) Soundness: S represents ground solutions to the query, i.e., $\bigcup_{s \in S} [\![s]\!] \subseteq S_{gnd}$, where $[\![\langle \Delta, G, D \rangle]\!] = \{\langle \Delta\theta, G\theta \rangle \mid \mathrm{ground}(\theta) \wedge D\theta = \mathrm{true}\}$ and S_{gnd} is the set of all ground solutions to the query, and (2) Comprehensiveness: S represents all minimal-residue ground solutions to the query, i.e., $\bigcup_{s \in S} [\![s]\!] \supseteq S_{min\text{-}gnd}$, where $S_{min\text{-}gnd}$ is the set of minimal-residue ground solutions to the query. Note that there may be multiple comprehensive solutions to a query.

Example. We illustrate abductive reachability queries and our analysis algorithm using the Treating Clinician policy presented in section 2.1, as a running example. Specifically, we take that policy as the main part of the initial policy P_0. Note that it is a fragment of the healthcare network policy case study in chapter 3. Additionally, we include the following facts about prototypical users in the initial policy P_0:

```
hasActivated(cli1, clinician(getWellHosp, surgeon)).
hasActivated(pat1, patient).
hasActivated(hpo1, policyOfficer(getWellHosp)).
```

These facts state that `cli1` is a prototypical surgeon at `getWellHosp`, `pat1` is a prototypical patient, and `hpo1` is a prototypical `getWellHosp` policy officer.

Informally, the desired query asks whether a clinician may be a treating clinician without having the patient's consent to treatment. To express this query, we also add the following rule to the initial policy:

```
treatingWithoutConsent(Pat, Cli)
:- memberOf(Cli, treatingClinician(Pat, getWellHosp)),
   ! consentToTreatment(Pat, Cli, getWellHosp)
```

`treatingWithoutConsent(Pat, Cli)` implies that `Cli` is a treating clinician for `Pat` without explicit consent to treatment for `Pat` at `getWellHosp`. With this rule in P_0, the goal in our reachability query can be expressed as `treatingWithoutConsent(pat1, cli1)`.

The entire initial policy P_0 and the query appear in Figures 4.1 and 4.2, respectively. The active administrators in the query are `hpo1` and `pat1`. In other words, we are asking whether the goal is reachabile through actions of `getWellHosp` policy officer and the patient mentioned in the goal. The set of abducible atoms include memberships in `getWellHosp` workgroups and encounters in the `getWellHosp`. It is reasonable to abduce these facts because the workgroup composition at a facility and encounter information are fluid and not known in advance. This allows us to obtain analysis solutions that are not specific to one particular configuration of workgroups and patient encounters.

Initial Policy P_0:

```
3.5.13
permit(User, addRule(memberOf(Cli, treatingClinician(Pat, getWellHosp))
                      :- consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(User, policyOfficer(getWellHosp))

3.5.12
permit(User, addRule(memberOf(Cli, treatingClinician(Pat, getWellHosp))
                      :- hasActivated(Cli, clinician(getWellHosp, Spcty)),
                         memberOf(Cli,
                                  workgroup(Wkgp, getWellHosp, Spcty, WkgpType)),
                         encounter(EncID, Pat, Wkgp, getWellHosp, Type)))
:- hasActivated(User, policyOfficer(getWellHosp))

3.5.7
permit(User, addRule(
               permit(Pat, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
               :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getWellHosp))

3.5.8
permit(User, addRule(
               permit(Pat, removeFact(consentToTreatment(Pat, Cli, getWellHosp)))
               :- hasActivated(Pat, patient)))
:- hasActivated(User, policyOfficer(getWellHosp))

3.5.9
permit(User, addRule(
               permit(Ag, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
               :- hasActivated(Ag, agent(Pat))))
:- hasActivated(User, policyOfficer(getWellHosp))

3.5.10
permit(User, addRule(
               permit(Ag, removeFact(consentToTreatment(Pat, Cli, getWellHosp)))
               :- hasActivated(Ag, agent(Pat))))
:- hasActivated(User, policyOfficer(getWellHosp))


hasActivated(cli1, clinician(getWellHosp, surgeon)).
hasActivated(pat1, patient).
hasActivated(hpo1, policyOfficer(getWellHosp)).
```

Figure 4.1: Initial policy P_0 for the all-solutions abductive reachability query example based on the Treating Clinician policy example presented in section 2.1 and based on the Healthcare Network case study presented in chapter 3.

- **Policy** P_0 : See Figure 4.1.
- **Set of active administrators** $U_0 = \{$`hpo1, pat1`$\}$
- **Goal** $G_0 =$ `treatingWithoutConsent(pat1, cli1)`
- **Specification of abducible atoms**: $A = \langle Ab, nAb \rangle$, where
  - Ab= $\{$`memberOf(User, workgroup(Wkgp, getWellHosp, Spcty, WkgpType)), encounter(EncID, Pat, Wkgp, getWellHosp, Type)`$\}$
  - nAb= $\{\}$

Figure 4.2: Example abductive atom-reachability query.

4.3 Becker *et al.*'s Algorithm for Tabled Policy Evaluation with Proof Construction and Abduction

This section briefly presents Becker *et al.*'s algorithm for tabled policy evaluation extended with proof construction and abduction [BN08, BMD09]. A modified version of their algorithm is used in our analysis algorithm, as described in the next section. This section is based closely on the presentation in [BMD09], however it also includes the extension for proof graph construction described in [BN08]. Their papers contain a more thorough exposition of this algorithm.

The algorithm appears in Figure 4.3. It is a deductive algorithm with abductive extension. The basic idea is this: during the resolution proof, whenever an attempt to prove a goal fails, the corresponding atom is nevertheless assumed to be true if the atom is abducible, in which case the atom is said to be *abduced*, and the proof continues. The algorithm keeps track of these assumptions, so each subgoal is associated with a set of abduced atoms on which its proof depends. The algorithm constructs a forest of proof trees. Each tree consists 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

```
**resolveClause** ($\langle P \rangle$)
$Ans(P) = \emptyset$
**for** $(Q \leftarrow \vec{Q}) \in Pol$
    **if** $nd = $ **resolve**$(\langle P; Q :: \vec{Q}; Q; []; Q \leftarrow \vec{Q}; \emptyset \rangle,$
                      $\langle P; []; P; []; \_; \emptyset \rangle)$ exists
        **processNode** $(nd)$
**if** $P$ is abducible
    **processAnswer** $(\langle P; []; P; []; abduction; [P] \rangle)$
```

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

```
**processNode** $(nd)$
**match** $nd$ **with** $\langle P; \vec{Q}; \_; \_; \_; \_ \rangle$ **in**
    **if** $\vec{Q} = []$
        **processAnswer** $(nd)$
    **else match** $\vec{Q}$ **with** $Q_0 :: \_$ **in**
        **if** there exists $Q_0' \in \mathtt{dom}(Ans)$
                such that $Q_0$ is an instance of $Q_0'$
            $Wait(Q_0') = Wait(Q_0') \cup \{nd\}$
            **for** $nd' \in Ans(Q_0')$
                **if** $nd'' = $ **resolve**$(nd, nd')$ exists
                    **processNode** $(nd'')$
        **else**
            $Wait(Q_0) = \{nd\}$
            **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',

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

Figure 4.3: Becker *et al.*'s deductive evaluation algorithm with abductive extension and proof construction

G that can be derived using the derivation that this node is part 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). [] represents an empty list and :: is the list constructor operator. Note that, in the definition of **resolveClause** in Figure 4.3, 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 $n = \langle G; \vec{Q}; S; \vec{c}; R; \Delta \rangle$, index$(n) = G$, subgoals$(n) = \vec{Q}$, pAns$(n) = S$, children$(n) = \vec{c}$, rule$(n) = R$, and residue$(n) = \Delta$.

An *answer table* is a partial function from atoms to sets of answer nodes. The set $Ans(G)$ contains all answer nodes for the goal G found so far.

A *wait table* is a partial function from atoms to sets of goal nodes. The set $Wait(G)$ 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 involving these waiting nodes is resumed.

The auxiliary definitions in the lower half of Figure 4.3 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 than n', so n can be discarded. **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 [BN08, BMD09], but the algorithm can handle them if the functions for matching and unification are extended appropriately. Becker *et al.* specify abducible atoms in a simpler way than we do, but this has no effect on the algorithm, other than the need to adopt our definition of "P is abducible", given in Section 4.2.

A clause in [BMD09] is a horn clause, or a rule in the policy (also called a program). Starting from some root node $\langle G \rangle$, resolution with program clauses produces goal nodes with index G. As the subgoals $\vec{Q}$ are processed one by one, new G-indexed goal nodes are created with the remaining subgoals and with increasingly instantiated variants of G as partial answer. A proof branch ends when no subgoals are left, that is, when an answer node is generated.

Invoking the Algorithm. The algorithm takes as input a query G, which is an atom, and the input policy *Pol*. The entry point is a call to **resolve-Clause** ($\langle G \rangle$). On termination, $Ans(G)$ contains a complete set of answers of the form $\langle G; []; S; \vec{c}; R; \Delta \rangle$, where S is a (not necessarily ground) instance of G. Such an answer can be interpreted as follows: if some ground instantiation of the atoms in Δ is added to the given policy *Pol*, then S, under the same ground instantiation, is derivable.

Correctness. [BN08] presents a soundness and a completeness argument for their algorithm. Their soundness theorem states that if the algorithm returns a proof graph for an answer node $\langle G; []; S; \vec{c}; R; \Delta \rangle$, then for all substitutions θ, such that $\Delta\theta$ is ground, $S\theta$ is subsumed by G and G is reachable from $Pol \cup \Delta\theta$. Their completeness theorem states that if an atom S is finite state reachable from an initial policy $Pol \cup A$, then there exists a substitution θ and an answer node $\langle G; []; S'; \vec{c}; R; \Delta \rangle$ that can be generated by algorithm for the query G, such that G subsumes S, $S'\theta = S$ and $\Delta\theta = A$. That is, in terms of completeness, this algorithm returns a most general set of solutions in terms of the subsumption relation. Note that the actual complete set of solutions may be infinite.

4.4 Analysis Algorithm

The algorithm has three phases. Phase 1 transforms the policy to eliminate `addRule` and `removeRule`. Phase 2 is a modified version of Becker *et al.*'s tabling algorithm described above; it produces candidate solutions. Recall that their algorithm attempts to derive a goal from a fixed policy. We modify the tabling algorithm, and transform its input, to enable it to compute sets of policy updates (i.e., administrative operations) needed to derive the goal. However, modifying the tabling algorithm to incorporate a notion of time (i.e., a notion of the order in which updates to the policy are performed, and of the resulting sequence of intermediate policies) would require extensive changes, so we do not do that. Instead, we introduce a third phase that 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.

4.4.1 Phase 1: Elimination of `addRule` and `removeRule`

The policy P' obtained by elimination of `addRule` and `removeRule` from a policy P is not completely equivalent to P—in particular, the state graphs $\mathrm{SG}(P, U_0)$ and $\mathrm{SG}(P', U_0)$ differ, and some kinds of properties, such as availability of permissions, are not preserved. However, P' is equivalent to P in the weaker sense that using P' in place of P in an abductive atom-reachability query does not change the answer to the query. Informally, this is because the restriction of negation to extensional relations implies that the answer to such a query depends only on the "upper bounds" of the derivable facts in reachable policies, not on the exact sets of derivable facts in each reachable policy, and this transformation preserves those upper bounds.

Elimination of `removeRule`. The policy $\mathrm{elimRmRule}(P)$ is obtained from P by simply deleting all `removeRule` permission rules (recall that policy safety, defined in Section 2.3, allows `removeRule` to appear only in the conclusion of such rules). This eliminates transitions that remove rules defining intensional predicates, and hence eliminates transitions that 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 reachable from P_0 is derivable in some policy reachable from $\mathrm{elimRmRule}(P_0)$. Conversely, since $\mathrm{SG}(\mathrm{elimRmRule}(P_0), U_0)$ is a subgraph of $\mathrm{SG}(P_0, U_0)$, every instance of the goal that is derivable in some policy reachable from $\mathrm{elimRmRule}(P_0)$ is derivable in some policy reachable from P_0. Therefore, the elimRmRule transformation does not affect the answer to abductive atom-reachability queries.

The initial policy P_0 in the treating clinician example does not contain any `removeRule` permission rules. Therefore, $\mathrm{elimRmRule}(P_0) = P_0$.

Elimination of `addRule`. We eliminate `addRule` by replacing `addRule` permission rules (recall that policy safety allows `addRule` to appear only in the conclusion of such rules) with new rules that use `addFact` to "simulate" the effect of `addRule`. Specifically, the policy $\mathrm{elimAddRule}(P)$ is obtained from P as follows. Let R be an `addRule` permission rule $\texttt{permit}(U, \texttt{addRule}$ $(L \texttt{ :- } \vec{L}_1)) \texttt{ :- } \vec{L}_2$ in P. Rule R is replaced with two rules. One rule is the rule pattern in the argument of `addRule`, extended with an additional premise using a fresh extensional predicate $\texttt{aux}_R$ that is unique to the rule: $L \texttt{ :- } \vec{L}_1, \texttt{aux}_R(\vec{X})$, where the vector of variables $\vec{X}$ is $\vec{X} = \mathrm{vars}(L \texttt{ :- } \vec{L}_1) \cap$

$(\mathrm{vars}(\{U\}) \cup \mathrm{vars}(\vec{L}_2))$. The other is an `addFact` permission rule that allows the user to add facts to this new predicate: $\texttt{permit}(U, \texttt{addFact}(\texttt{aux}_R(\vec{X})))$ `:-` $\vec{L}_2$. The auxiliary predicate $\texttt{aux}_R$ keeps track of which instances of the rule pattern have been added. Recall from Section 2.3 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 $\mathrm{vars}(\{U\}) \cup \mathrm{vars}(\vec{L}_2)$, 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 $\texttt{aux}_R$ records the values of those variables. In other words, an `addRule` transition t in $\mathrm{SG}(P_0, U_0)$ in which the user adds an instance of the rule pattern with $\vec{X}$ instantiated with $\vec{c}$ is "simulated" in $\mathrm{SG}(\mathrm{elimAddRule}(P_0), U_0)$ by an `addFact` transition t that adds $\texttt{aux}_R(\vec{c})$.

Note that $\mathrm{SG}(P_0, U_0)$ 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 adding premises to it. Those transitions are eliminated by this transformation, in other words, there are no corresponding transitions in $\mathrm{SG}(\mathrm{elimAddRule}(P_0), U_0)$. This is sound, because those transitions lead to policies in which the intensional predicate p that appears in literal L (i.e., L is $p(\ldots)$) 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.

Applying this transformation to a policy satisfying the fixed administrative policy requirement produces a policy containing no higher-order administrative permission rules.

Example. For example, adding a fact to the auxiliary predicate $\texttt{aux}_{3.5.7}()$ simulates adding an `addFact` permission rule using `addRule` permission rule 3.5.7. Note that a nullary predicate may contain no facts of it may contain a single fact represented by the 0-tuple ().

Therefore, by adding a fact for the predicate $\texttt{aux}_{3.5.7}()$, `policyOfficer( getWellHosp)` can "activate" the rule with this predicate in the premises.

Figure 4.4 presents the complete policy $\mathrm{elimAddRule}(P_0)$ for the initial policy P_0 in the treating clinician example from Figure 4.1.

Correctness. This section presents the correctness arguments in more detail.

First, we show that reachability of atoms is preserved when `addRule`

```
memberOf(Cli, treatingClinician(Pat, getWellHosp))
:- consentToTreatment(Pat, Cli, getWellHosp), aux_{3.5.13}()

permit(User, addFact(aux_{3.5.13}()))
:- hasActivated(User, policyOfficer(getWellHosp))

memberOf(Cli, treatingClinician(Pat, getWellHosp))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   memberOf(Cli, workgroup(Wkgp, getWellHosp, Spcty, WkgpType)),
   encounter(EncID, Pat, Wkgp, getWellHosp, Type), aux_{3.5.12}()

permit(User, addFact(aux_{3.5.12}()))
:- hasActivated(User, policyOfficer(getWellHosp))

permit(Pat, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Pat, patient), aux_{3.5.7}()

permit(User, addFact(aux_{3.5.7}()))
:- hasActivated(User, policyOfficer(getWellHosp))

permit(Pat, removeFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Pat, patient), aux_{3.5.8}()

permit(User, addFact(aux_{3.5.8}()))
:- hasActivated(User, policyOfficer(getWellHosp))

permit(Ag, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Ag, agent(Pat)), aux_{3.5.9}()

permit(User, addFact(aux_{3.5.9}()))
:- hasActivated(User, policyOfficer(getWellHosp))

permit(Ag, removeFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Ag, agent(Pat)), aux_{3.5.10}()

permit(User, addFact(aux_{3.5.10}()))
:- hasActivated(User, policyOfficer(getWellHosp))

hasActivated(cli1, clinician(getWellHosp, surgeon)).
hasActivated(pat1, patient).
hasActivated(hpo1, policyOfficer(getWellHosp)).
```

Figure 4.4: elimAddRule(elimRmRule(P_0)) for the policy P_0 presented in Figure 4.1.

transitions are restricted so they don't add premises and when `removeRule` transitions are eliminated.

The *restricted transition relation* T_- is defined the same way as the transition relation T in Section 2.6 except (1) `addRule` transitions are restricted so that they do not add additional premises to rule patterns, and (2) `removeRule` transitions are eliminated.

The *restricted state graph* $\mathrm{SG}_-(P_0, U_0)$ for policy P_0 and set U_0 of users is defined in the same way as the state graph for P_0 and U_0, except using the restricted transition relation T_- instead of the transition relation T.

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'.

Lemma 4.4.1. *For all policies P and P', if $P \leq P'$, then $[\![P]\!] \subseteq [\![P']\!]$.*

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 its less-or-equally-strict version R' in P' (in other words, R' is the rule 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'. To see this, note that R' is less-or-equally-strict than R, so 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 $\texttt{!}a$, then a must be an atom for an extensional predicate, so it suffices to consider the facts that appear explicitly in P and P'. 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'. □

Theorem 4.4.2. *For every policy P_0 and set U_0 of users, for every policy P in $\mathrm{SG}(P_0, U_0)$, there exists a policy P' in $\mathrm{SG}_-(P_0, U_0)$ such that $P \leq P'$.*

Proof. Let $p_i = P_0 \overset{u_0:op_0}{\longrightarrow} P_1 \overset{u_1:op_1}{\longrightarrow} \dots \overset{u_{n-1}:op_{n-1}}{\longrightarrow} P_n$ be a path in $\mathrm{SG}(U_0, P_0)$ from the initial policy to P, hence $P_n = P$. We show by construction that there is a corresponding path $p'_i = P_0 \overset{u'_0:op'_0}{\longrightarrow} P'_1 \overset{u'_1:op'_1}{\longrightarrow} \dots \overset{u'_{n-1}:op'_{n-1}}{\longrightarrow} P'_n$ in $\mathrm{SG}_-(P_0, U_0)$ 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`$(u_i, \texttt{addRule}(R'_i))$ using R_i^{arp} in the last step of the derivation, and R_i is stricter than R'_i. (note: R'_i already reflects instantiations of variables.)

p'_i is defined as follows.

$$u'_i = u_i$$

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

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

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`$(u'_0, op'_0) \in [\![P_0]\!]$.

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`$(u'_0, op'_0) \in [\![P_0]\!]$ follows directly from `permit`$(u_0, op_0) \in [\![P_0]\!]$.

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`$(u_0, \texttt{addRule}(R'_0)) \in [\![P_0]\!]$.

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

(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`$(u'_{i+1}, op'_{i+1}) \in [\![P'_{i+1}]\!]$.

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 $\texttt{permit}(u'_{i+1}, op'_{i+1}) \in [\![P'_{i+1}]\!]$ follows directly from $\texttt{permit}(u_{i+1}, op_{i+1}) \in [\![P_{i+1}]\!]$, $P_{i+1} \leq P'_{i+1}$, and Lemma 4.4.1.

case: op_{i+1} is $\texttt{addRule}(R_{i+1})$. In this case, op'_{i+1} is $\texttt{addRule}(R'_{i+1})$, as defined above. The definition of R'_i in that paragraph directly implies that $\texttt{permit}(u_{i+1}, \texttt{addRule}(R'_{i+1})) \in [\![P_{i+1}]\!]$. This, together with $P_{i+1} \leq P'_{i+1}$ and Lemma 4.4.1, imply that $\texttt{permit}(u_{i+1}, \texttt{addRule}(R'_{i+1})) \in [\![P'_{i+1}]\!]$.

□

Theorem 4.4.3. *For every policy P_0, set U_0 of users, and atom a,* $\mathrm{SG}(P_0, U_0)$ *contains a policy P with $a \in [\![P]\!]$ iff* $\mathrm{SG}_-(P_0, U_0)$ *contains a policy P' with $a \in [\![P']\!]$.*

Proof. We prove one direction at time.

Suppose $\mathrm{SG}(P_0, U_0)$ contains a policy P such that $a \in [\![P]\!]$. We need to show that there exists P' in $\mathrm{SG}_-(P_0, U_0)$ such that $a \in [\![P']\!]$. This follows immediately from Theorem 4.4.2, the definition of $\leq$, Lemma 4.4.1, and the fact that goals do not contain negative intensional literals.

Suppose $\mathrm{SG}_-(P_0, U_0)$ contains a policy P' such that $a \in [\![P']\!]$. We need to show that $\mathrm{SG}(P_0, U_0)$ contains a policy P such that $a \in [\![P]\!]$. This follows from the fact that the restricted transition relation T_- is a subset of the transition relation T, which implies that $\mathrm{SG}(P_0, U_0)$ also contains P. □

Next, we show that the restricted state graph for the original policy is similar to the (full) state graph for the transformed policy, i.e., $\mathrm{SG}_-(P_0, U_0)$ is similar to $\mathrm{SG}(\mathrm{elimAddRule}(\mathrm{elimRmRule}(P_0)), U_0)$ in the sense defined below. We call predicates with names like $\texttt{aux}_R$ as *auxiliary predicates.* We assume that the original policy does not contain auxiliary predicates.

Policies P and P' are similar, denoted $P \simeq P'$, if P and P' contain the same rules and facts with three exceptions: (1) P' contains no `removeRule` permission rules, (2) P contains no rules involving auxiliary predicates, P' contains no `addRule` permission rules, 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 elimAddRule transformation, and (3) for every `addRule` permission rule $\texttt{permit}(U, \texttt{addRule}(L \texttt{ :- } \vec{L}_1)) \texttt{ :- } \vec{L}_2$ in P, for every ground substitution θ, P contains $(L \texttt{ :- } \vec{L}_1)\theta$ iff either P'

contains $(L \texttt{ :- } \vec{L}_1)\theta$ or P' contains $\texttt{aux}_R(\vec{X})\theta$ and $(L \texttt{ :- } \vec{L}_1, \texttt{aux}_R(\vec{X}))$, where $\vec{X} = \text{vars}(L \texttt{ :- } \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$.

Informally, (3) says that the facts that have been added to auxiliary predicates in P' are exactly the facts needed to simulate the rules that have been added to P.

Let excludedAtoms denote the set of atoms of the form $\texttt{permit}(\ldots, \texttt{addRule}(\ldots))$, $\texttt{permit}(\ldots, \texttt{removeRule}(\ldots))$, $\texttt{aux}_R(\ldots)$, or $\texttt{permit}(\ldots, \texttt{addFact}(\texttt{aux}_R(\ldots)))$.

Lemma 4.4.4. *If $P \simeq P'$, and $a \notin$ excludedAtoms, then $a \in [\![P]\!]$ iff $a \in [\![P']\!]$.*

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 predicate. By hypothesis, $a \notin$ excludedAtoms, so a is not a fact for an auxiliary predicate. Therefore, $a \in [\![P]\!]$ iff $a \in [\![P']\!]$.

case a is intensional: First we consider the forward direction of the "iff", i.e., we assume $a \in [\![P]\!]$ and show $a \in [\![P']\!]$. Let D be a derivation of a using facts and rules in P. Without loss of generality, we assume D does not contain uses of `removeRule` permission rules; since negation cannot be applied to intensional relations, such a derivation exists for every derivable fact. 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 R used in D. If there exists an `addRule` permission rule $\texttt{permit}(U, \texttt{addRule}(L \texttt{ :- } \vec{L}_1)) \texttt{ :- } \vec{L}_2$ in P, such that R is $(L \texttt{ :- } \vec{L}_1)\theta$ for some ground substitution θ, then by item (3) in the definition of $\simeq$, P' contains either the same rule R or an auxiliary fact and transformed rule R' that can be used to derive the same conclusion; in the latter case, we replace the use of R in D with a use of R'. R is not an `addRule` permission rule, because $a \notin$ excludedAtoms (so the top-level rule in D is not an `addRule` permission rule) and the definition of ACAR in Chapter 2) does not allow `permit` to appear in premises of rules (so `addRule` permission rules are not used to derive any subgoals in D). R is not a `removeRule` permission rule, by the assumption above. Thus, for any other rule R used in D, the definition of $\simeq$ implies that R also exists in P'. The definition of $\simeq$ implies that each fact in P that is used in D also exists in P'. Therefore, D' is a derivation of a in P'.

Now we consider the reverse direction of the "iff", i.e., we assume $a \in [\![P']\!]$ and show $a \in [\![P]\!]$. 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 R' used in D'. If one of the premises of R' involves an auxiliary predicate, then item (3) in the definition of $\simeq$ implies that P contains a rule R that can be used to derive the same conclusion, so we replace the use of R' in D' with a use of R. By reasoning similar to that in the previous paragraph, every other rule and every fact used in D' also exists in P. Therefore, D is a derivation of a in P.

□

Theorem 4.4.5. *(1) For every policy P, $P \simeq \text{elimAddRule}(\text{elimRmRule}(P))$. (2) 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 $\langle P, u{:}op, P_1\rangle \in T_-$, there exists a policy P_1' and operation op' such that $\langle P', u{:}op', P_1'\rangle \in T$ and $P_1 \simeq P_1'$, and (b) for every policy P_1' and operation op' such that $\langle P', u{:}op', P_1'\rangle \in T$, there exists a policy P_1 and operation op such that $\langle P, u{:}op, P_1\rangle \in T_-$ and $P_1 \simeq P_1'$.*

Proof. (1) This follows from the definitions of elimAddRule and elimRmRule functions and $\simeq$ relation.

(2) (a) Note that $\langle P, u{:}op, P_1\rangle \in T_-$ implies that `permit`$(u, op) \in [\![P]\!]$. We perform a case analysis on the kind of administrative operation that op is.

case op is `removeRule`: T_- does not contain `removeRule` transitions, so this case cannot occur.

case op is `addFact` or `removeFact` for a non-auxiliary predicate: In this case, `permit`(u, op) is not in excludedAtoms, so `permit`$(u, op) \in [\![P]\!]$ and $P \simeq P'$ imply, using Lemma 4.4.4, that `permit`$(u, op) \in [\![P']\!]$, 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 $\langle P', u{:}op', P_1'\rangle \in T$ and $P_1 \simeq P_1'$.

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

case op is `removeFact` for an auxiliary predicate: Item (2) in the definition of $\simeq$ implies that P and P' do not contain `removeFact` rules for auxiliary predicates, so this case cannot occur.

case *op* is `addRule`: $\texttt{permit}(u, op) \in [\![P]\!]$ implies there is an `addRule` permission rule R of the form $\texttt{permit}(U, \texttt{addRule}(L \texttt{ :- } \vec{L}_1)) \texttt{ :- } L_2$ used with a ground substitution θ to derive $\texttt{permit}(u, op)$ in P; thus, $u = U\theta$, and $op = \texttt{addRule}(L \texttt{ :- } \vec{L}_1)\theta$. Item (2) in the definition of $\simeq$, and the definition of elimAddRule, together imply that P' contains the rules $L \texttt{ :- } \vec{L}_1, \texttt{aux}_R(\vec{X})$ and $\texttt{permit}(U, \texttt{addFact}(\texttt{aux}_R(\vec{X}))) \texttt{ :- } \vec{L}_2$, where $\vec{X} = \text{vars}(L \texttt{ :- } \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$. Let R' denote the latter rule. We take op' to be $\texttt{addFact}(\texttt{aux}_R(\vec{X}))\theta$, and P'_1 to be the policy obtained by executing op' in P'. To see that $\texttt{permit}(u, op') \in [\![P']\!]$, and hence $\langle P', u{:}op', P'_1\rangle \in T$, note that the premises of R' are the same as the premises of R, and they cannot be excluded atoms (because the definition of ACAR in Chapter 2 does not allow permissions as premises, and the definition of $\simeq$ implies that P and hence R do not contain auxiliary predicates), and these premises are derivable in P, so Lemma 4.4.4 implies that they are derivable in P'. It is straightforward to show that $P_1 \simeq P'_1$.

(b) Note that $\langle P', u{:}op', P'_1\rangle \in T$ implies that $\texttt{permit}(u, op') \in [\![P']\!]$. We perform a case analysis on the kind of administrative operation that op' is.

case op' is `removeRule`: $P \simeq P'$ implies that there are no `removeRule` permission rules in P', so this case cannot occur.

case op' is `addFact` or `removeFact` for a non-auxiliary predicate: In this case, $\texttt{permit}(u, op')$ is not in excludedAtoms, so $\texttt{permit}(u, op) \in [\![P']\!]$ and $P \simeq P'$ imply, using Lemma 4.4.4, that $\texttt{permit}(u, op) \in [\![P]\!]$, 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 $\langle P, u{:}op'\, P'_1\rangle \in T_{-}$ and $P_1 \simeq P'_1$.

case op' is `addRule`: $P \simeq P'$ implies that there are no `addRule` permission rules in P', so this case cannot occur.

case op' is `addFact` for an auxiliary predicate: $\texttt{permit}(u, op) \in [\![P']\!]$ implies there is an `addFact` permission rule R of the form $\texttt{permit}(U, \texttt{addFact}(\texttt{aux}_R(\vec{X}))) \texttt{ :- } \vec{L}_2$ used with a ground substitution θ to derive $\texttt{permit}(u, op')$ in P'; thus, $u = U\theta$, and $op' = \texttt{addFact}(\texttt{aux}_R(\vec{X}))\theta$. Item (2) in the definition of $\simeq$, and the definition of elimAddRule, together imply that P' also contains the rule $L \texttt{ :- } \vec{L}_1, \texttt{aux}_R(\vec{X})$, where $\vec{X} = \text{vars}(L \texttt{ :- } \vec{L}_1) \cap (\text{vars}(\{U\}) \cup \text{vars}(\vec{L}_2))$, and P contains the `addRule` permission rule $\texttt{permit}(U, \texttt{addRule}(L \texttt{ :- } \vec{L}_1)) \texttt{ :- } L_2$. Let R denote the latter rule. We take op to be $\texttt{addRule}(L \texttt{ :- } \vec{L}_1)\theta$, and P_1 to be the policy obtained by executing op in P. To see that $\texttt{permit}(u, op) \in [\![P]\!]$, and hence $\langle P, u{:}op, P_1\rangle \in T_-$, note that the premises of R are the same as the premises of R', and they cannot be excluded atoms (because the definition of ACAR in Chapter 2 does not allow permissions as premises, and the definition of $\simeq$ implies that R was produced by the elimAddRule transformation and hence $\vec{L}_2$ is copied from the original policy and does not contain auxiliary predicates), and these premises are derivable in P, so Lemma 4.4.4 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: Item (2) in the definition of $\simeq$ implies that P and P' do not contain `removeFact` rules for auxiliary predicates, so this case cannot occur.

□

Theorem 4.4.6. *For every policy P_0, set U_0 of users, and atom $a \notin$ excludedAtoms, $\mathrm{SG}_-(P_0, U_0)$ contains a policy P with $a \in [\![P]\!]$ iff $\mathrm{SG}(\mathrm{elimAddRule}(\mathrm{elimRmRule}(P_0)), U_0)$ contains a policy P' with $a \in [\![P']\!]$.*

Proof. The proof for this theorem follows from the lemma for path correspondence for bisimilar states in [EMCGP99], which says that if s and s' are two bimilar states such then for every path from s there is a corresponding path starting from s', and vice versa. Theorem 4.4.5 establishes that $\simeq$ is a bisimulation relation between P_0 and elimAddRule(elimRmRule(P_0)). Therefore, for every policy P_0 and set U_0 of users, if $P \in \mathrm{SG}_-(P_0, U_0)$ then there exists $P' \in \mathrm{SG}((\mathrm{elimAddRule}(\mathrm{elimRmRule}(P_0)), U_0)$ such that $P \simeq P'$, and vice versa. Therefore, from Lemma 4.4.4, for such policies P and P', $a \in [\![P]\!]$ iff $a \in [\![P']\!]$. □

Putting these pieces together, we conclude that the transformations to eliminate `addRule` and `removeRule` preserve atom-reachability.

Theorem 4.4.7. For every policy P_0, set U_0 of users, and atom a not in excludedAtoms, $\mathrm{SG}(\ \mathrm{elimAddRule}(\mathrm{elimRmRule}(P_0)), U_0)$ contains a policy P' with $a \in [\![P']\!]$ iff $\mathrm{SG}(P_0, U_0)$ contains a policy P with $a \in [\![P]\!]$.

With this theorem, it is easy to show that answers to abductive atom-reachability queries are preserved by this transformation. Therefore, subsequent phases of the analysis algorithm analyze the transformed policy elimAddRule(elimRmRule(P_0)).

4.4.2 Phase 2: Tabled Policy Evaluation

Phase 2 is a modified version of Becker *et al.*'s algorithm presented in Section 4.3. Phase 2 considers three ways to satisfy a positive subgoal: through an inference rule, through addition of a fact (using an `addFact` permission rule), 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).

To allow the algorithm to explore addition of facts as a way to satisfy positive subgoals, without directly modifying the algorithm, we transform `addFact` permission rules into ordinary inference rules. Specifically, each `addFact` permission rule $\texttt{permit}(U, \texttt{addFact}(a))$ `:-` $\vec{L}$ is replaced with the rule a `:-` $\vec{L}, \texttt{u0}(U)$. The transformation also introduces a new extensional predicate `u0` and, for each $u \in U_0$, the fact $\texttt{u0}(u)$ is added to the policy. For example, consider the following `addFact` permission rule in the transformed policy elimAddRule(elimRmRule(P_0)) in Figure 4.4:

```
permit(Pat, addFact(consentToTreatment(Pat, Cli, getWellHosp)))
:- hasActivated(Pat, patient), aux3.5.7()
```

This rule is replaced with the rule:

```
consentToTreatment(Pat, Cli, getWellHosp)
:- hasActivated(Pat, patient), aux3.5.7(), u0(Pat)
```

The set of active administrators $U_0 = \{\texttt{hpo1, pat1}\}$ is represented as facts `u0(hpo1), u0(pat1)` in the transformed policy.

This transformation changes the meaning of the policy: the transformed rule means that a necessarily holds when $\vec{L}$ holds, while the original `addFact` permission rule means that a might (or might not) be added by an administrator when $\vec{L}$ holds. This difference is significant if a appears negatively in a premise of some rule. This change in meaning is acceptable in phase 2, because phase 2 does not attempt to detect conflicts between negative subgoals and added facts. As discussed in Section **??**, this change in the meanings of rules used in phase 2 does not affect the detection of such conflicts in phase 3.

The algorithm considers two ways to satisfy a negative subgoal: through removal of a fact (using a `removeFact` permission rule) 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).

To allow the algorithm to explore removal of facts as a way to satisfy negative subgoals, `removeFact` permission rules are transformed into ordinary inference rules with negative conclusions. Specifically, each `removeFact` permission rule `permit`$(U,$ `removeFact`$(a))$ `:-` $\vec{L}$ is replaced with the rule $!a$ `:-` $\vec{L},$ `u0`(U).

Let elimAddRmFact(P) denote the policy obtained from P by transforming `addFact` and `removeFact` rules as described above. An *administrative node* (or "admin node", for short) is a node n such that rule(n) is a transformed `addFact` or `removeFact` permission rule. isAdmin(n) holds iff n is an administrative node. isAddFact(n) holds iff rule(n) is a transformed `addFact` permission rule. isRmFact(n) holds iff rule(n) is a transformed `removeFact` permission rule. Figure 4.5 presents the policy obtained by applying elimAddRmFact to the policy in Figure 4.4.

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 "$P \in [\![A]\!] \vee (\exists a \in \text{Atom}_{ex}.\ a \notin P_0 \wedge P$ is $!a)$", where Atom_{ex} is the set of extensional atoms. If a is not ground, disequalities in d_{init} in phase 3 will ensure that the solution includes only instances of a that are not in P_0.

Note that wildcards do not need any special treatment in phase 2. To establish a negative premise that contains wildcards. To establish the premise using abduction, the negative literal is simply abduced (with wildcards in it) into the residue. This might lead to disequalities that contain wildcards, specifically, disequalities of the form $\_ \neq t$, where t does not contain wildcards (because wildcards cannot appear as arguments of constructors). The function satisfiable(d) used in Phase 3 to test satisfiability of a disequality d handles wildcards as follows: a disequality of the form $\_ \neq t$ is not satisfiable. Recall from Section 2.8 that wildcards can be used in a negative literal $!p(\ldots)$ only if there are no `removeFact` permission rules for p. This means that we do not need to consider how to establish negative literals containing wildcards using removals of facts. Trying to establish such a premise using removals is difficult, because we cannot precisely determine, in phase 2, what instances of the predicate need to be removed. Extending the algorithm to handle this is future work and a brief summary of a scheme to do so has been described later in section 6.2.

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 3 interprets "!" as negation and checks appropriate consistency conditions by detecting inconsistencies between positive and negative facts. Phase 3 is also responsible for detecting if a `removeFact` operation (corresponding to a use of a transformed `removeFact` rule in phase 2) falsifies a subsequent positive subgoal.

```
treatingWithoutConsent(Pat, Cli)
:- memberOf(Cli, treatingClinician(Pat, getWellHosp)),
   !consentToTreatment(Pat, Cli, getWellHosp)

memberOf(Cli, treatingClinician(Pat, getWellHosp))
:- consentToTreatment(Pat, Cli, getWellHosp), aux_{3.5.13}()

aux_{3.5.13}() :- u0(User), hasActivated(User, policyOfficer(getWellHosp))

memberOf(Cli, treatingClinician(Pat, getWellHosp))
:- hasActivated(Cli, clinician(getWellHosp, Spcty)),
   memberOf(Cli, workgroup(Wkgp, getWellHosp, Spcty, WkgpType)),
   encounter(EncID, Pat, Wkgp, getWellHosp, Type), aux_{3.5.12}()

aux_{3.5.12}() :- u0(User), hasActivated(User, policyOfficer(getWellHosp))

consentToTreatment(hpo1, Cli, getWellHosp)
:- u0(Pat), hasActivated(Pat, patient), aux_{3.5.7}()

aux_{3.5.7}() :- u0(User), hasActivated(User, policyOfficer(getWellHosp))

!consentToTreatment(hpo1, Cli, getWellHosp)
:- u0(Pat), hasActivated(Pat, patient), aux_{3.5.8}()

aux_{3.5.8}() :- u0(User), hasActivated(User, policyOfficer(getWellHosp))

consentToTreatment(Pat, Cli, getWellHosp)
:- u0(Ag), hasActivated(Ag, agent(Pat)), aux_{3.5.9}()

aux_{3.5.9}() :- u0(User), hasActivated(User, policyOfficer(getWellHosp))

!consentToTreatment(Pat, Cli, getWellHosp)
:- u0(Ag), hasActivated(Ag, agent(Pat)), aux_{3.5.10}()

aux_{3.5.10}() :- u0(User), hasActivated(User, policyOfficer(getWellHosp))

u0(hpo1).
u0(pat1).

hasActivated(cli1, clinician(getWellHosp, surgeon)).
hasActivated(pat1, patient).
hasActivated(hpo1, policyOfficer(getWellHosp)).
```

Figure 4.5: Transformed policy obtained after applying elimAddRmFact transformation to the policy elimAddRule(elimRmRule(P_0)) in Figure 4.4.

Although phase 3 is responsible for checking satisfaction of negative subgoals, phase 2 cannot simply ignore negative subgoals, because if a `removeFact` permission rule needs to be used to satisfy a negative subgoal, phase 2 is responsible for exploring ways to satisfy the premises of that rule.

The definition of **resolve** in Figure 4.3 checks whether **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 $[\![Ab]\!]$ to $[\![nAb]\!]$, causing it not to be abducible. Therefore, in the definition of **resolve**, we replace the condition **unifiable**(Q, Q'') with the condition **unifiable**$(Q, Q'') \wedge (\Delta\theta \cup \Delta''\theta \subseteq [\![A]\!])$. Note that it is sufficient to consider only instantiation with the most general unifier, not less general unifiers, because nAb is closed under instantiation.

Becker *et al.* algorithm explores all derivations for a goal except that the subsumption check in **processAnswer** in Figure 4.3 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 3, 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 means equality modulo renaming of variables that do not appear in the 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, because, if n' could be used as part of a larger derivation that is classified as feasible in phase 3, then n could be used instead of n', because the constraints generated in phase 3 when n is used are satisfiable whenever the constraints generated when n' is used are satisfiable, because the additional nodes in the proof of n' just introduce additional constraints. 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 **unifiable**(Q, Q'') with **unifiable**$(Q, Q'') \wedge \text{noCyclicDeriv}(n')$, where

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

where the *proof* of a node n, denoted 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 isAns$(n)$ holds iff n is an answer node. 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 rule(n') and subgoals(n').

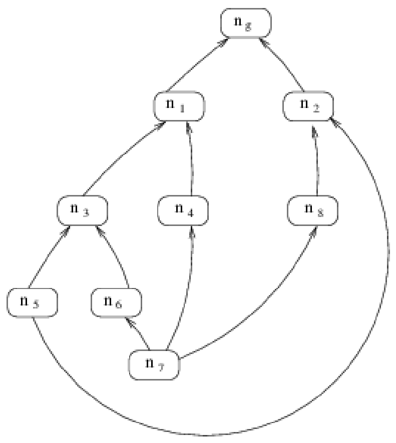
We extend the algorithm to store the *partial answer substitution*, denoted $\theta_{\text{pa}}(n)$, in each node n. This is the substitution that, when applied to index(n), produces pAns(n) (the pAns component is therefore redundant and could be eliminated). In the **resolveClause** method, the θ_{pa} 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, given an abductive atom-reachability query of the form in Section 4.2, phase 2 applies the tabling algorithm, modified as described above, to the policy elimAddRmFact(elimAddRule(elimRmRule(P_0))) with the given goal G_0 and specification A of abducible atoms.

Example. Figures 4.6 and 4.7 present two proof graphs ψ_1 and ψ_2 generated for the example query in Figures 4.1 and 4.2. In both the graphs, n_g represents an answer node for the goal $G_0 =$ `treatingWithoutConsent(pat1, cli1)`. A directed edge (n, n') in a proof graph implies that n is an answer node for a subgoal resulting from the rule used to derive node n'.

4.4.3 Phase 3: Ordering Constraints

Phase 3 considers constraints on the execution order of administrative operations. The ordering must ensure that, for each administrative node or goal node n, (a) 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 (b) each assumption about the initial policy on



(a) Proof graph ψ_1

n_g	index = treatingWithoutConsent(pat1, cli1)
	pAns = treatingWithoutConsent(pat1, cli1)
	residue = {}

n_1	index = memberOf(cli1, treatingClinician(pat1, `getWellHosp`))
	pAns = memberOf(cli1, treatingClinician(pat1, `getWellHosp`))
	residue = {}

n_2	index = !consentToTreatment(pat1, cli1, `getWellHosp`)
	pAns = !consentToTreatment(pat1, cli1, `getWellHosp`)
	residue = {}

n_3	index = consentToTreatment(pat1, cli1, `getWellHosp`)
	pAns = consentToTreatment(pat1, cli1, `getWellHosp`)
	residue = {}

n_4	index = $\text{aux}_{3.5.13}()$
	pAns = $\text{aux}_{3.5.13}()$
	residue = {}

n_5	index = hasActivated(pat1, patient)
	pAns = hasActivated(pat1, patient)
	residue = {}

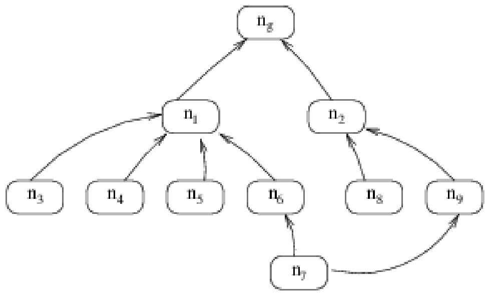
n_6	index = $\text{aux}_{3.5.7}()$
	pAns = $\text{aux}_{3.5.7}()$
	residue = {}

n_7	index = hasActivated(hpo1, policyOfficer(`getWellHosp`))
	pAns = hasActivated(hpo1, policyOfficer(`getWellHosp`))
	residue = {}

n_8	index = $\text{aux}_{3.5.8}()$
	pAns = $\text{aux}_{3.5.8}()$
	residue = {}

(b) Answer nodes

Figure 4.6: First proof graph ψ_1 generated in phase 2 for the example query in Figure 4.1 and 4.2 using the transformed policy elimAddRmFact(elimAddRule(elimRmRule(P_0))) in Figure 4.5.



(a) Proof graph ψ_2

n_g	index = treatingWithoutConsent(pat1, cli1)
	pAns = treatingWithoutConsent(pat1, cli1)
	residue = {memberOf(cli1, workgroup(Wkgp, `getWellHosp`, Spcty, WkgpType)), encounter(EncID, pat1, Wkgp, `getWellHosp`, Type)}

n_1	index = memberOf(cli1, treatingClinician(pat1, `getWellHosp`))
	pAns = memberOf(cli1, treatingClinician(pat1, `getWellHosp`))
	residue = {memberOf(cli1, workgroup(Wkgp, `getWellHosp`, Spcty, WkgpType)), encounter(EncID, pat1, Wkgp, `getWellHosp`, Type)}

n_2	index = !consentToTreatment(pat1, cli1, `getWellHosp`)
	pAns = !consentToTreatment(pat1, cli1, `getWellHosp`)
	residue = {}

n_3	index = hasActivated(cli1, clinician(`getWellHosp`, Spcty))
	pAns = hasActivated(cli1, clinician(`getWellHosp`, Spcty))
	residue = {}

n_4	index = memberOf(cli1, workgroup(Wkgp, `getWellHosp`, Spcty, WkgpType))
	pAns = memberOf(cli1, workgroup(Wkgp, `getWellHosp`, Spcty, WkgpType))
	residue = {memberOf(cli1, workgroup(Wkgp, `getWellHosp`, Spcty, WkgpType))}

n_5	index = encounter(EncID, pat1, Wkgp, `getWellHosp`, Type)
	pAns = encounter(EncID, pat1, Wkgp, `getWellHosp`, Type)
	residue = {encounter(EncID, pat1, Wkgp, `getWellHosp`, Type)}

n_6	index = $\text{aux}_{3.5.12}$()
	pAns = $\text{aux}_{3.5.12}$()
	residue = {}

n_7	index = hasActivated(hpo1, policyOfficer(`getWellHosp`))
	pAns = hasActivated(hpo1, policyOfficer(`getWellHosp`))
	residue = {}

n_8	index = hasActivated(pat1, patient)
	pAns = hasActivated(pat1, patient)
	residue = {}

n_9	index = $\text{aux}_{3.5.8}$()
	pAns = $\text{aux}_{3.5.8}$()
	residue = {}

(b) Answer nodes

Figure 4.7: Second proof graph ψ_2 generated in phase 2 for the example query in Figure 4.1 and 4.2 using the transformed policy elimAddRmFact(elimAddRule(elimRmRule(P_0))) in Figure 4.5.

which n relies is not undone by an operation that occurs before n (this is an "interference-freedom constraint").

Note that, when generating the ordering constraints in item (a) 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 (a) is considered for node n'.

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.

Pseudocode for phase 3 appears in Figures 4.8 and 4.9. The algorithm generates the overall constraint, puts the Boolean expression in disjunctive normal form (DNF), and then checks, for each clause c, whether the generated ordering constraints 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, and includes the disequalities that label the removed ordering constraints in the solution. After constraints have been checked (including the 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 (note that 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—roughly speaking, the supersets not containing the negative literals). Additional disequalities are also added to each solution (d_{nAb}) that reflect that the residue must be disjoint from nAb.

Example. Figures 4.10 and 4.11 present the overall ordering constraint and the conjunctive clauses produced by the orderingConstraints function in Figure 4.9 for the proof graph ψ_1 in Figure 4.6. The cycles in each clause are highlighted by thicker lines. Note that none of the cycles can be removed while maintaining satisfiability of the resulting constraint. Therefore, proof

```
$solutions = \emptyset$
for each node $n_g \in Ans(G)$
    // consistency constraint: disequalities that ensure consistency of initial state,
    // i.e., positive literals are distinct from negative literals.
    $d_{\text{init}} = \bigwedge\{\text{args}(a) \neq \text{args}(b) \mid a \in \text{facts}(P_0) \cup \text{residue}(n_g) \wedge !b \in \text{residue}(n_g) \wedge \textbf{unifiable}(a, b)\}$
    $d_{nAb} = \bigwedge\{\text{args}(a) \neq \text{args}(b) \mid a \in \text{residue}(n_g) \wedge b \in nAb \wedge \textbf{unifiable}(a, b)\}$
    $d_0 = d_{nAb} \wedge d_{\text{init}}$
    if $\neg$satisfiable($d_0$)
      continue
    endif
    $O$ = orderingConstraints($n_g$)
    if ($\exists$ clause $c$ in $O$. 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.
        $solutions = solutions \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d_0\rangle\}$
    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 each clause $c$ in $O$
            if mightNeedRepeatedOp($c, n_g$)
              // the current version of the algorithm does not support repeated operations
              return "repeated operations might be needed"
            endif
            $D_{\text{ord}} = \emptyset$
            // $c$ 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 $D_{\text{ord}}$ 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 $D_{\text{ord}}$.
            // we use the algorithm in [RND77] to compute $Cyc$.
            $Cyc$ = set of all cycles in ordering constraints for clause $c$
            $FAS = \{F \mid F$ contains one edge selected from each cycle in $Cyc\}$
            // $smFAS$ is the set of $\subseteq$-minimal feedback arc sets (FASs) for clause $c$
            $smFAS = \{F \in FAS \mid \nexists F' \in FAS.\ F' \subset F\}$
            for each $F$ in $smFAS$
              $d = d_0 \wedge \bigwedge\{d' \mid \langle n_1, n_2, d'\rangle \in F\}$
              if satisfiable($d$) $\wedge \neg(\exists d' \in D_{\text{ord}}.\ d' \subseteq d)$
                $D_{\text{ord}} = D_{\text{ord}} \cup \{d\}$
              endif
            endfor
            $solutions = solutions \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d\rangle \mid d \in D_{\text{ord}}\}$
        endfor
    endif
endfor
// return solutions that are not subsumed by other solutions
return $\{s \in solutions \mid \neg\exists s' \in solutions.\ s \preceq_S s'\}$
```

Figure 4.8: Pseudo-code for Phase 3.

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

$\text{args}(a)$ = a tuple containing the arguments of atom a

$\text{support}(n) = \{n' \in \text{proof}(n) \mid \text{isAns}(n') \wedge n' \neq n$
$\neg\exists n_a.\text{isAdmin}(n_a) \wedge \text{descendant}(n, n_a) \wedge \text{descendant}(n_a, n')\}$

$\text{adminSupport}(n) = \{n' \in \text{support}(n) \mid \text{isAdmin}(n')\}$

$\text{supportingInitFact}(n) = \{\text{pAns}(n') \mid n' \in \text{support}(n)$
$\wedge\, (\text{rule}(n') \in \text{facts}(P_0) \vee \text{rule}(n') = \text{abduction})\}$

Figure 4.9: Ordering constraints for an answer node n_g.

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

Figure 4.10: DNF representation of ordering constraint O_{ψ_1} generated by phase 3 function orderingConstraints for node n_g in the proof graph ψ_1 in Figure 4.6. f abbreviates "*false*".

graph ψ_1 does not yield any satisfiable solutions for the abductive reachability query in Figures 4.1 and 4.2. Figures 4.12 and 4.13 present the ordering constraint and the conjunctive clause produced by the orderingConstraints function for node n_g in the proof graph ψ_2 in Figure 4.7. There is only one clause, and it is acyclic. Therefore, ψ_2 yields a candidate solution to the query.

Removal of subsumed solutions. Another consequence of replacing the subsumption check in **processAnswer** with an equality check is that phase 2 may produce solutions that are subsumed by other solutions. This is permitted by the definition of the abductive reachability problem in Section 4.2 but is undesirable nevertheless. Therefore, we define a subsumption relation on solutions, and remove subsumed solutions from *solutions* near the end of the pseudocode in Figure 4.8.

Informally, a solution $\langle \Delta, G, D\rangle$ is subsumed by ("is less general than") a solution $\langle \Delta', G', D'\rangle$ if Δ' contains fewer or more general atoms than Δ, G' is more general than G, and D' contains fewer tuple disequalities than D. Formally, a solution $\langle \Delta, G, D\rangle$ is *subsumed by* a solution $\langle \Delta', G', D'\rangle$, denoted $\langle \Delta, G, D\rangle \preceq_S \langle \Delta', G', D'\rangle$, iff $|\Delta| \geq |\Delta'|$ and there exists a substitution θ such that $G = G'\theta$ and $\Delta \supseteq \Delta'\theta$ and $D \supseteq D'\theta$. It is easy to show that, if $s \preceq_S s'$, then $[\![s]\!] \subseteq [\![s']\!]$.

Optimization. As an optimization, we implement a heuristic that can eliminate processing of some answer nodes n_g in the main loop of phase

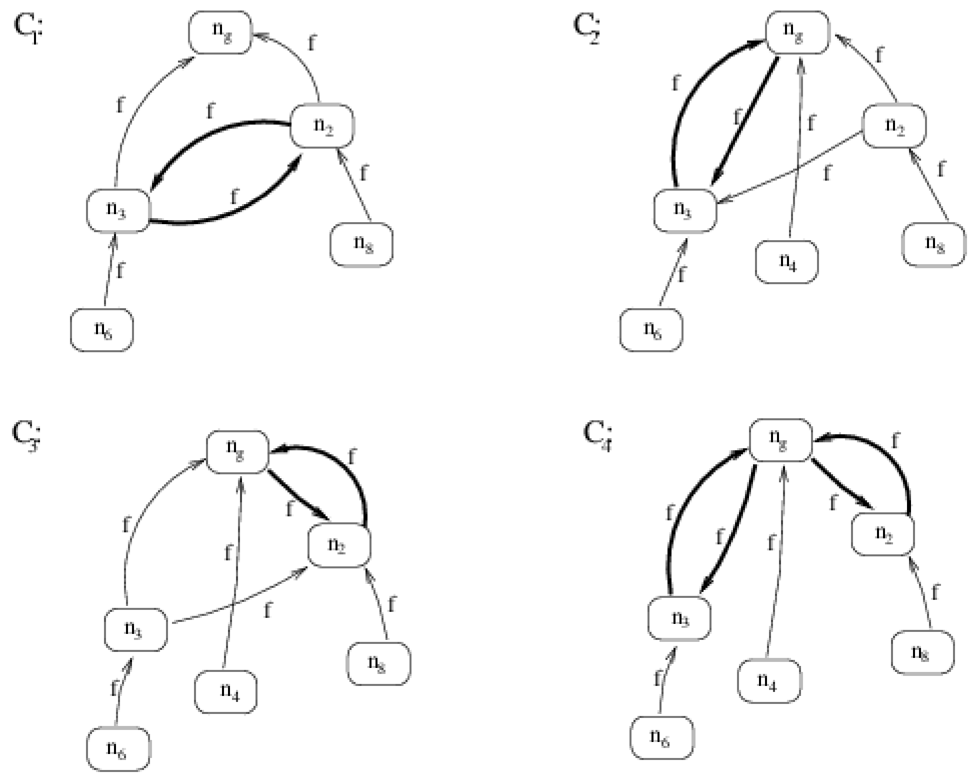


Figure 4.11: Conjunctive clauses in O_{ψ_1} from Figure 4.10 represented as labeled graphs. The unsatisfiable cycles are highlighted by thicker lines.

3 in Figure 4.8. Before starting the main loop, we construct a directed acyclic graph (DAG) of answer nodes with an edge from n to n' if $n \preceq_{node} n'$, where the subsumption relation $\preceq_{node}$ on answer nodes is defined by: $n \preceq_{node} n'$ iff $|\text{residue}(n)| \geq |\text{residue}(n')|$ and there exists a substitution θ such that $\text{index}(n) = \text{index}(n')\theta$ and $\text{pAns}(n) = \text{pAns}(n')\theta$ and $\text{residue}(n) \supseteq \text{residue}(n')\theta$. After processing a node n_g in the main loop of phase 3 in Figure 4.8, if n_g leads to a solution with no disequalities (i.e., the third component of the tuple is *true*) then we discard all nodes n' that are reachable from n_g in the DAG, because it is easy to prove, from the definition of $\preceq_{node}$ and $\preceq_S$, that processing of node n' would only yield solutions (if any) that are subsumed (in the sense of $\preceq_S$) by the solutions yielded by processing of n_g (and hence if we computed the solutions from n', they would be discarded in the last line of Figure 4.8).

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

Figure 4.12: DNF representation of ordering constraint O_{ψ_2} generated by phase 3 function orderingConstraints for node n_g in the proof graph ψ_2 in Figure 4.7. f abbreviates "*false*".

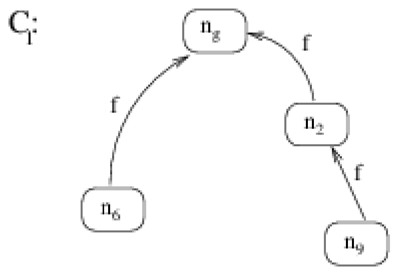


Figure 4.13: Conjunctive clause in O_{ψ_2} from Figure 4.12 represented as labeled graphs. The ordering constraint is acyclic.

Repeated Administrative Operations. Tabling re-uses nodes, including, in our setting, 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 current version of our algorithm 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 (see Figure 4.8). Specifically, mightNeedRepeatedOp(c, n_g) returns true if some node n in c is a child of multiple nodes in proof(n_g); in such cases, it might be necessary to replace n with multiple nodes, one for each parent, in order to satisfy the ordering constraints. To achieve this, the algorithm can be modified so that, if mightNeedRepeatedOp returns true, the algorithm reruns phases 2 and 3 but this time constructs new answer nodes, instead of re-using tabled answers, for the nodes identified by mightNeedRepeatedOp as possibly needing to be repeated.

Plan Construction. We describe how to extend the algorithm to return plans, i.e., sequences of administrative actions that lead to a policy in which

```
function nodeToAction($n$)
  if(isAddFact($n$)) then
    $op = \texttt{addFact}(\text{pAns}(n))$
  else
    $op = \texttt{removeFact}(\text{pAns}(n))$
  $n'$ = the node in children($n$) such that $\textbf{unifiable}(\text{pAns}(n'), \texttt{u0}(U))$
  $\theta = \textbf{mostGeneralUnifier}(\text{pAns}(n'), \texttt{u0}(U))$
  $u = U\theta$
  return $u{:}op$
```

Figure 4.14: nodeToAction function to compute actions corresponding to an administrative answer node

an instance of the goal is derivable.

The algorithm is modified to return a set of 4-tuples $\langle \Delta, G, D, \Pi \rangle$ where Δ, G, and D are as before, and Π is a set of sequences of administrative actions $u : op$ (not necessarily ground), such that for every sequence π in Π and every ground substitution θ such that satisfiable($D\theta$), the sequence of actions $\pi\theta$ can be executed starting from $P_0 \cup \Delta\theta$ and leads to a policy P such that $G_0\theta \in [\![P]\!]$.

Each administrative node n in a proof graph represents an administrative action. The plan construction algorithm uses the nodeToAction function in Figure 4.14, which takes as input an administrative node n and returns an administrative action $u{:}op$ corresponding to that node.

The pseudo-code in Figure 4.8 is modified as follows to construct plans. The line

$$\textit{solutions} = \textit{solutions} \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d_0 \rangle\}$$

is replaced with

$$\textit{solutions} = \textit{solutions} \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d_0, \text{plans}(n_g, c) \rangle\}$$

where the function plans(n_g, c) is defined by

```
function plans($n_g, c$)
  $A = \{n \in \text{proof}(n_g) \mid \text{isAdmin}(n)\}$
  $\Pi_{\text{node}} = \{\pi \in \text{Perm}(A) \mid$
            $\forall n, n' \in A : ((n, n', \ldots) \in c) \Rightarrow n$ precedes $n'$ in $A\}$
  return $\{\text{map}(\text{nodeToAction}, \pi) \mid \pi \in \Pi_{\text{node}}\}$
```

where Perm(A) is the set of permutations of A, and map(f, o) maps the function f along the sequence o and returns the sequence containing the results. Also, the line

$$D_{\text{ord}} = D_{\text{ord}} \cup \{d\}$$

is replaced with

$$D_{\text{ord}} = D_{\text{ord}} \cup \{\langle d, F\rangle\}$$

and the expression $\exists d' \in D_{\text{ord}}$ is replaced with $\exists\langle d', F'\rangle \in D_{\text{ord}}$, and the line

$$\mathit{solutions} = \mathit{solutions} \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d\rangle \mid d \in D_{\text{ord}}\}$$

is replaced with

$$\mathit{solutions} = \mathit{solutions} \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d, \text{plans}(n_g, c \setminus F\rangle \mid \langle d, F\rangle \in D_{\text{ord}}\}$$

4.5 Correctness

The algorithm is sound in the sense that, when it terminates with a solution, the solution is indeed a comprehensive solution to the given abductive atom-reachability query. A soundness proof appears below.

The algorithm is incomplete, for two reasons. First, it might diverge, for policies of the kind for which Becker *et al.*'s algorithm might diverge [BN08, Section 3.2] and for other policies as well, due to replacement of the subsumption check in **processAnswer** with an alpha-equality check. Second, it might return the message that repetition of administrative actions might be needed, as discussed in Section 4.4.3, instead of returning a solution.

Soundness of phases 2 and 3 in the extended algorithm are expressed by the following theorem.

Theorem 4.5.1. *Let P_0' denote the policy obtained by applying* elimAddRule *and* elimRmRule *transformations to the given initial policy P_0 in phase 1. Suppose the extended phase 3 terminates and returns a set S. Let $\langle\Delta, G, D, \Pi\rangle$ be an element of S, and let π be an element of* Π. *For every ground substitution θ such that* satisfiable$(D\theta)$*, the sequence of actions $\pi\theta$ can be executed starting from $P_0' \cup \Delta\theta$ and leads to a policy P such that $G\theta \in [\![P]\!]$.*

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 $0 \le i < |\pi - 1|$,

$\pi_i\theta$ is executable in P'_i, i.e., there exists a (unique) policy P'_{i+1} such that $\langle P'_i, \pi_i\theta, P'_{i+1}\rangle \in T$.

Base case: Every first action π_0 in any plan returned by the algorithm is allowed under the corresponding initial policy $P'_0 \cup \Delta\theta$. This is because, since we only consider addition and removal of facts, the inference rules are immutable and can be used anywhere for derivation and, from the construction of derivations for phase 2, the action $\pi_0\theta$ corresponds to an admin node in the proof graph that depends only on the facts and rules in the initial policy and is, therefore, derivable under the corresponding initial policy $P'_0 \cup \Delta\theta$.

Step case: Suppose n_{i+1} is the node in $\text{proof}(n_g)$ such that $\pi_{i+1} = \text{nodeToAction}(n_{i+1})$. From the proof graph construction in phase 2, we know that n_{i+1} can be proven using $n_0, ..., n_i$ and additional nodes which correspond to derivations in the proof graph and answer nodes that use abduction. Since phase 1 eliminated addition and removal of rules, phases 2 and 3 consider only addition and removal of facts, so the inference rules are immutable and can be used anywhere for derivation. Also, the answer nodes that abduce facts for the initial policy are appropriately ordered with respect to the administrative nodes that require them, due to the ordering constraints in $O_{\text{add-init}}$ and $O_{\text{rm-init}}$. The elimAddRmFact transformation used in phase 2 changes the semantics of the policy in the following way: facts that were not derivable in the untransformed policy, because they were not added to the policy by an administrator, are derivable through the inference rules produced by this transformation. However, these facts could be added by an administrator, so this consequence of the transformation does not compromise soundness. Furthermore, phase 2 does not interpret the negation symbol as negation, so it allows inconsistent derivations to be generated as a result of using the transformed rules. However, Phases 2 and 3 together are sound, because phase 3 eliminates inconsistent derivations generated in phase 2 because it does not use the transformed rules; instead, phase 3 re-interprets uses of the transformed inference rules in the proof graph as `addFact` and `removeFact` operations, constructs ordering constraints on those operations, and checks for inconsistencies in the intermediate and final policies produced by performing those operations in an order consistent with the ordering constraints.

The requirement that the plan construction is consistent with a satisfiable ordering constraint O, and the assumption satisfiable($D\theta$), ensures that $\pi_{i+1}\theta$ can be executed after $\pi_0\theta, \ldots, \pi_i\theta$, i.e., that $\langle P'_i, \pi_i\theta, P'_{i+1}\rangle \in T$.

Thus, for all substitution θ such that satisfiable($D\theta$), the plan π is sound with respect to the transition relation T. Since, the plan construction stops when a policy is reached that derives G, the final action $\pi_f\theta \in \pi\theta$ results in the policy P such that $P \vdash G\theta$.

□

Comprehensiveness of phases 2 and 3 in the extended algorithm are expressed by the following theorem. It states that if the algorithm terminates and returns a solution then all minimal ground solutions to the transformed policy are instances of some solution returned by the algorithm.

Theorem 4.5.2. *Let P'_0 denote the policy obtained by applying* elimAddRule *and* elimRmRule *transformations to the given initial policy P_0 in phase 1. Suppose, there exists a set of ground facts Δ_m such that there exists a sequence of administrative actions A from $P'_0 \cup \Delta_m$ to P and $P \vdash G_m$ where G_m is a ground instance of G_0 and for all $\Delta' \subset \Delta_m$, there does not exist any ground substitution ν such that $G_0\nu$ is reachable from $P_0 \cup \Delta'$. Then there exists $\langle \Delta, G, D, \Pi\rangle \in S$ such that for some substitution θ, $\Delta\theta = \Delta_m$ and G_m is an instance of $G\theta$.*

Proof. elimAddRmFact transformation changes the semantics of the policy P'_0 such that facts that were not derivable in the untransformed policy, because they were not added to the policy by an administrator, are derivable through the new inference rules. These changes, however, do not cause phase 2 to overlook any possible derivations of the goal. This follows from the fact that phase 2 ignores the semantics of negation (recall that it treats the negation symbol as part of the predicate name) and hence may construct proof graphs that contain both an atom and its negation. In effect, the transformed policy used in phase 2 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 not to be considered. Of course, this also means that some of the generated derivations might be inconsistent, i.e., they might contain both a and $!a$. Since, Becker *et al.*'s tabling algorithm is complete and our extension to the algorithm only increases the semantics of the transformed policy in phase 2, phase 2 is comprehensive in terms of preserving the minimal solutions to the input query.

We argue that phase 3 does not eliminate the minimal solutions to the query. This is because, if there does exist a feasible sequence of actions as represented by the administrative nodes in one of the proof graphs from phase 2 (which will be preserved as argued earlier) then this would be explored while considering all possible plans in the plan construction and the resulting disequality D would be satisfiable for the corresponding minimal ground solution. Thus, there would exist a solution $\langle \Delta, G, D, \Pi \rangle \in S$ such that for some substitution θ, $\Delta\theta = \Delta_m$ and G_m is an instance of $G\theta$, where $\langle \Delta_m, G_m \rangle$ is the minimal ground solution in consideration. □

Overall Correctness of the Algorithm. Next we state the correctness of the overall algorithm when it terminates and returns a set of solutions.

Theorem 4.5.3. *Let $Q = \langle P_0, U_0, A, G_0 \rangle$ be an abductive atom reachability query. Suppose the algorithm extended with plan construction terminates and returns a set S. Then S is a comprehensive solution to Q, that is:*

Soundness: $\bigcup_{s \in S} [\![s]\!] \subseteq S_{gnd}$, *where* $[\![\langle \Delta, G, D, \Pi \rangle]\!] = \{\langle \Delta\theta, G\theta \rangle \mid \text{ground}(\theta) \wedge D\theta = \text{true}\}$ *and S_{gnd} is the set of all ground solutions to Q, and*

Comprehensiveness: $\bigcup_{s \in S} [\![s]\!] \supseteq S_{min\text{-}gnd}$, *where* $[\![\langle \Delta, G, D, \Pi \rangle]\!] = \{\langle \Delta\theta, G\theta \rangle \mid \text{ground}(\theta) \wedge D\theta = \text{true}\}$ *and $S_{min\text{-}gnd}$ is the set of minimal-residue ground solutions to Q.*

Proof. **Soundness:** From theorem 4.4.7 we know that the atom-reachability for all atoms $a \notin \text{excludedAtoms}$ is preserved under phase 1 of the algorithm. Also, from theorem 4.5.1, we know that all solutions returned from phases 2 and 3 are sound, i.e. they $\forall s \in S. [\![s]\!] \subseteq S_{gnd}$. Therefore, $\bigcup_{s \in S} [\![s]\!] \subseteq S_{gnd}$.

Comprehensiveness: Phase 1 of the algorithm preserves comprehensiveness because atom-reachability is preserved as per theorem 4.4.7. Further, from theorem 4.5.2, we know that, if the algorithm terminates with a solution, all minimal ground solutions to Q are instances of some solution returned by the algorithm. Therefore, if the overall algorithm returns a set S, then $\forall s \in S_{min\text{-}gnd}(\exists s' \in S(s \in [\![s']\!])) \Rightarrow \bigcup_{s \in S} [\![s]\!] \supseteq S_{min\text{-}gnd}$.

□

4.6 Implementation and Experience.

We implemented the analysis algorithm in approximately 5000 lines of OCaml. The choice of framework for the implementation was motivated by the strong need for pattern matching and abstract syntax tree manipulation in the program. We applied the implementation to part of the policy P_{HCN} for the healthcare network case study in chapter 3 with 23 administrative permission rules. We included facts about a few prototypical users in P_{HCN}: `fpo1`, a member of `policyOfficer(getWellHosp)`; `clin1`, a clinician at `getWellHosp`; and `user1`, a user with no roles. A sample abductive atom-reachability query that we evaluated has $P_0 = P_{\mathrm{HCN}}$, $U_0 = \{$`fpo1`, `user1`$\}$, $A = \langle Ab = \{$`memberOf(User, workgroup(WG, getWellHosp, Spcty, team))`$\}$, $nAb = \{\}\rangle$, and $G_0 =$ `workgroupHead(GoalUser, cardioTeam, getWellHosp)`. The analysis takes about 3 seconds, generates 2352 nodes, and returns five solutions. For example, one solution has partial answer `workgroupHead(GoalUser, cardioTeam, getWellHosp)`, residue $\{$`memberOf(GoalUser, workgroup(cardioTeam, getWellHosp, Spcty, team))`$\}$, and tuple disequality $\langle$`GoalUser`$\rangle \neq \langle$`fpo1`$\rangle$. The disequality reflects that `fpo1` can appoint himself to the `hrManager(getWellHosp)` 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, due to the negative premise in the rule 3.5.1.

Another query that we have tested and evaluated the implementation for included the prototypical users `fpo1`, a member of `policyOfficer(getWellHosp)`; `hhr`, a member of `hrManager(getWellHosp)`; `cli1` a clinician at `getWellHosp`; and `pat1`, a patient at `getWellHosp`. In this case, $U_0 = \{$`fpo1`, `hhr`$\}$, $A = \langle Ab = \{$ `memberOf(User, workgroup(WG, getWellHosp, Spcty, team)), encounter(EncID, pat1, Wkgp, getWellHosp, Type)`$\}$, $nAb = \{\}\rangle$, and $G_0 =$ `memberOf(cli1, treatingClinician(pat1, getWellHosp))`. To summarize, the query asks whether through action of the policy officer and HR manager, can a clinician become the treating clinician for a patient at `getWellHosp`, i.e. without the patient or the clinician's involvement. The analysis takes about 4 seconds and returns one solution with residue $\{$`memberOf(cli1, workgroup(WG, getWellHosp, Spcty, team), encounter(EncID, pat1, WG, getWellHosp, Type)`$\}$ which indicates that this is possible if `hhr` makes `cli1` a member of a workgroup `WG` that is currently handling an encounter `EncID` for `pat1`. This may not be a malicious behavior, but from the perspective of a policy auditor, an understanding of such a scenario helps in identifying behaviors that were not intended during policy design.

4.7 Related Work

Several analysis algorithms for determining user-permission reachability in ARBAC97 [SBM99] have been proposed. [LT06] proposes state-exploration based security analysis techniques as a means to maintain security properties during delegation of administrative privileges.

[JLT$^+$08] considers various classes of security analysis problems for RBAC which ask whether an access control system preserves security policy invariants across policy state changes. An important goal in [JLT$^+$08] is to help RBAC administrators precisely understand whom they are trusting for maintaining the desirable security properties, that is, who will be able to compromise the security of their system. They present two approaches, one based on model checking, and the other based on logic programming.

[SYRG07, SYGR11, SYSR11] address the user-role reachability problem in ARBAC, which asks whether a given user can be assigned to given roles by given administrators.

User-role reachability for ARBAC is intractable in general. [SYRG07] presents an algorithm for user-role reachability analysis for ARBAC and expresses its worst-case time complexity in tersm of parameters that characterize the hardness of the problem instance. They show that, when these parameters are fixed, the time complexity of the algorithm is polynomial in terms of the overall input size. They also argue that the hardness parameters are often small in practice.

[SYGR11] extends ARBAC to parameterized ARBAC (PARBAC), which adds parameters to roles and permissions, thereby enhancing the scalability, flexibility and expressiveness of ARBAC. It also presents a proof that user-role reachability for PARBAC is undecidable in case of infinite domains for parameters and presents a semi-decision procedure for reachability analysis of PARBAC.

[SYSR11] addresses the worst-case time complexity for a wide range of ARBAC analysis problems. It presents analysis for reachability, availability (e.g., whether a user cannot be removed from a role by a group of administrators), containment (e.g., every member of one role is also a member of another role), and information flow properties in ARBAC. It also shows that reachability analysis for ARBAC is PSPACE-complete and then considers the affect on the worst-case complexity of various restrictions on the problem instances.

The administrative frameworks and analysis algorithms in all of these works on ARBAC differ significanlty from our work in that the administrative operations they consider correspond to addition and removal of facts, based on the representation of RBAC in ACAR in Section 3.3; thus, they do not consider administrative operations that correspond to addition and removal of rules.

[BHA11] extends UARBAC, an RBAC administration model proposed in [LM07], to improve expressiveness and usability and proposes logic and program verification techniques to solve a reachability problem intended to aid legitimate users in understanding how to achieve a desired access-control state. While their extended version of UARBAC is significantly more expressive than ARBAC97, all policies under their framework can also be expressed in ACAR by translating their definitions for action permissions to ACAR rules. Note that UARBAC *create* operation, which creates a new object of a specified class, can be modeled in ACAR as an `addFact` operation that adds a fact to a relation representing the extent of the class. UARBAC is less expressive than ACAR, because it does not support rules, permissions to add or remove rules, negative preconditions for actions, etc. The analysis algorithm in [BHA11] allows more general goals than our analysis algorithm, with disjunction, conjunction, and negation, with the restriction that negation cannot occur in the scope of conjunction. On the other hand, their analysis algorithm does not handle the above features of ACAR, and it does not perform abductive analysis.

[KN07] proposes a decidable safety analysis problem for the HRU access control model [HRU76]. In HRU, a policy is represented by a set of commands, which correspond roughly to administrative permission rules in ACAR. The general safety (reachability) analysis problem for HRU asks whether a safety property (e.g., unreachability of a specified goal) holds for a specified policy starting from a specified initial state (represented as an access matrix, corresponding roughly to initial facts in ACAR). This problem is undecidable in [HRU76]. [KN07] identify a decidable variant of the safety problem, which asks whether a temporal safety property holds for a specified policy starting from all initial states. This is reminiscent of abductive analysis, because the initial state is unspecified; on the other hand, it is not abductive analysis, because it does not compute conditions on the initial state that ensure the safety property holds. Also, our definition of abductive reachability allows the user to specify part or all of the initial facts if desired; for example, the user can completely specify the initial facts by including them in P_0 and taking the set of abducibles to be empty.

Craven *et al.* [CLM$^+$09] present policy analysis for dynamic authorization policies. They present a rich authorization language with support for obligations, time constraints, stratified negation (which we don't), etc. They motivate the use of abductive logic programming to answer policy analysis questions as it presents more information in an answer than simply yes and no. They present a set of policy analysis problems that can be addressed using abductive logic programming, such as policy comparison, which is a comparison of two policies for equivalence or subsumption, separation of duty, etc. However, their policy language itself does not include a rule-based administrative model. On the other hand, one may be able to model addition and removal of facts as events in their system, which can be controlled by the policy itself and hence serve as an administrative policy. Another limitation in their work, particularly in the analysis algorithm, is that they restrict abduction to ground residues. On the other hand, our algorithm tries to address a more general problem of computing comprehensive solutions and computes non-ground residues.

There is little work on administrative frameworks for rule-based access control. [Bec09, BN10] present a framework for administration of rule-based access control, however they consider only addition and removal of facts, not rules. [Bec09] presents a dynamic authorization policy language and two reachability analysis approaches for this language to present plans of actions (addition and removal of authorization facts) that lead to the goal. The first method is based on AI planning, however it requires the domain of constants in the language to be finite and the policy to be *tight* meaning that every rule defining an intensional predicate could be finitely unfolded down to extensional predicates. The second method, based on theorem proving, relaxes the finiteness restriction but makes the problem undecidable. However, as stated earlier, both approaches only deal 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.

Chapter 5

Verification of Security Policy Enforcement in Enterprise Systems

An important problem in security policy research is verification of policy enforcement to ensure that, given the overall system architecture and security configurations of various system components, a *high-level* security policy is correctly enforced. Such high-level policies can be used to express security requirements for entire systems and may refer to abstract information resources, independent of where the information is stored or how the storage is implemented. Further, such policies control both direct and indirect access requests to the information and may refer to the request's path through the system (what we refer to as the request *context*). Such high-level policies are enforced several security mechanisms and are distributed across different system components. The overall system architecture may also affect the policy enforcement by restricting the paths that an access request may take. Ensuring that the individual components' security configurations (*low-level* policy) and the system architecture correctly enforce a high-level policy is a difficult problem. This chapter presents our research on this problem. Also, enforcement of the high-level policies might involve multiple hardware and software components in the system. Therefore, a natural question during security analysis is to identify a *trusted computing base* (TCB) for each information resource. The answer to such a question may depend on the low-level policies as well as the system architecture.

In this chapter we first explicitly identify the characteristics of high-level policies. Next we present a framework that allows *convenient* and *formal* specification of such high-level policies, modeling of low-level policies, and modeling of relevant aspects of system architecture. Finally, we describe a method for verifying that the low-level policies in a system correctly enforce ("implement") the high-level policies and an algorithm for computing a trusted computing base for a component or information resource.

5.1 Framework

5.1.1 Running Example.

We use a student information system as a running example to illustrate our framework. Student information is classified as academic (transcript, etc.) or personal (SSN, citizenship, etc.). The system architecture is shown in Figure 5.1. Academic information and personal information are stored in separate databases. `solar` is a web-based university information system; for brevity, we model `solar` and the associated web server as a single component.

5.1.2 Information Resources.

An *information resource*, abbreviated IR, represents a kind of information handled by the system. The relation `implements(`C`, `I`)` means that component C (partially or completely) implements IR I, i.e., C stores that kind of information. For example, the student information system contains two IRs, `academicIR` and `personalIR`, each implemented by a corresponding database (e.g., `implements (academicDB, academicIR)`). The distinction between an IR and the components that implement it is useful if the information in the IR is partitioned, replicated, archived, etc.

The information in an IR is assumed to be structured as a set of records, whose attributes (fields) and their types are specified in the definition of the IR. We refer to these as attributes of the IR, although they are actually attributes of the records in it. An attribute type can be a primitive data type (e.g., String) or an IR, denoting a reference to a record in another IR (recursive types are prohibited). For example, the attributes of `academicIR` and `studentIR` include an attribute `id` with type String, which identifies the student that the record is about. IRs have a straightforward API with operations for manipulating records. For example, the API includes an operation `readField` with arguments `record` (the record being accessed) and `field` (the field being accessed).

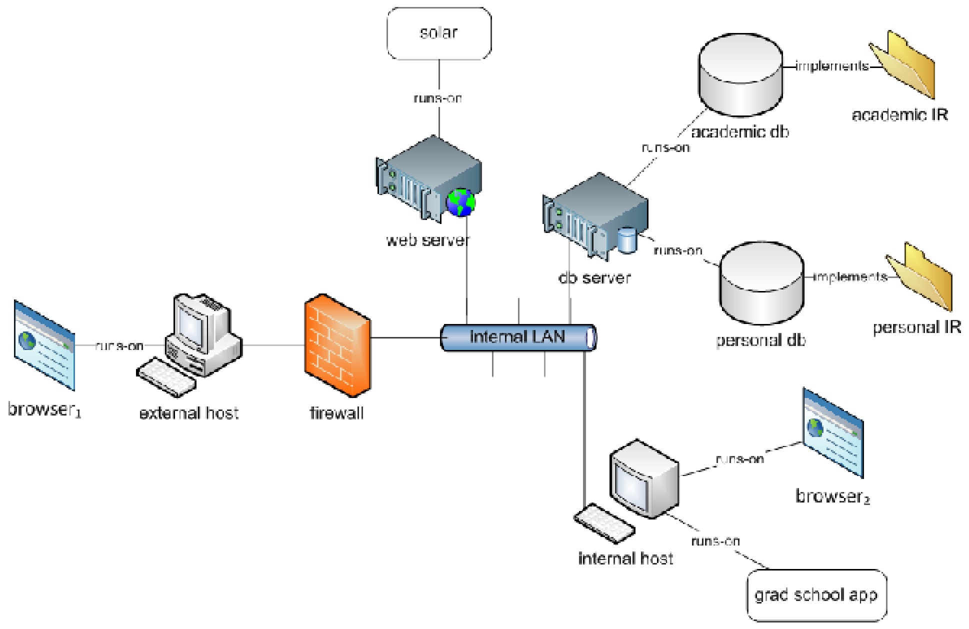


Figure 5.1: Architecture of student information system. Edge labels specify the corresponding relation. The components connected on `internal LAN` are related to each other via `link` relation.

5.1.3 Components.

A system is built from *components*, which may represent software (e.g., `solar`) or hardware (e.g., a host or firewall). Each component has attributes, accessed using the dot operator. For example, for a software component C, C`.host` is the host on which C runs. Attributes can also provide information about identity management, e.g., which authentication services and directory services are used by the component.

Each component has an API. For example, the API for the databases `academicDB` and `personalDB` is modeled (ignoring details of SQL) as containing functions like `readField`, `writeField`, `readRecord`, and `addRecord`. The API for `solar` contains `getTranscript`, `getSSN`, and `getCitizenship`. We model the browser as offering its user a single function, `request`, which non-deterministically sends some request to a web server (in this case, solar). For brevity, we consider only the above functions; other functions can be modeled and analyzed similarly.

Each component has a *low-level permit policy* that controls invocations of functions in the component's API and is enforced locally by the component. The language for low-level policies is described later in this section.

5.1.4 High-Level Policies.

High-level policies are expressed in a simple rule-based language, which is an extension of Datalog with simple data structures that can be read, but not constructed or updated, by policy rules. A policy rule has the form Q `<-` P_1`,...,`P_n and means: Q holds if P_1 through P_n hold. Variables start with an uppercase letter, constants start with a lowercase letter, and string constants appear in single quotes. The rules define the relation `hPermit` ("high-level permit"). `hPermit(`U`, `R`, `Op`, `C`)` holds if the system should permit (allow) requests from user U to perform operation Op on resource R in context C. A *resource* is a component or IR. The rules may also define auxiliary relations. For convenience, the name and arguments of the operation are modeled as attributes of Op (this is just a modeling convention, not an assumption about the implementation); the operation name is stored in Op`.function`. The context C is a sequence of tuples (c, f)—where c is a component or IR, and f is a function in c's API—representing the call chain (or "path") by which the request propagated through the system. Figure 5.2 shows some high-level policies for the running example.

```
% A Student can read any field in the records for himself or
% herself.
(P1) hPermit(User, Resource, Op, Context) <-
       Resource in {academicIR, personalIR},
       Op.function = readField, Op.record.id = User.id

% A Graduate School Clerk can read every student’s transcript,
% if accessed through solar from (a browser running on) an
% internal host. Note: Context.head() is the first element of
% the context. internalHost(H) is an auxiliary predicate
% (definition elided) that holds if host H is part of the campus
% network.
(P2) hPermit(User, academicIR, Op, Context) <-
      Op.function = readField, Op.field = ’transcript’,
      User.role = ‘GradSchlClerk’, Context.contains(solar),
      runs-on(Context.head(), H), internalHost(H)

% A registrar can read a student’s personal information, if
% accessed from an internal host
(P3) hPermit(User, personalIR, Op, Context) <-
      Op.function = readRecord, User.role = ’Registrar’,
      runs-on(Context.head(), H), internalHost(H)

% An administrative user can add new records to academicIR
(P4) hPermit(User, academicIR, Op, Context) <-
      Op.function = addRecord, User.role = ‘admin’

% An administrative user can add new records to personalIR
(P5) hPermit(User, personalIR, Op, Context) <-
      Op.function = addRecord, User.role = ‘admin’
```

Figure 5.2: Illustrative high-level policy rules for the student information system

5.1.5 Call Map

A function in a component's API may call functions provided by other components. Such calls must be considered to determine whether the restrictions on indirect calls expressed by high-level policies are enforced. We introduce a function `callMap` that captures the possible calls made by each component function. For simplicity and efficiency, `callMap` provides, and our analysis tracks, only equalities involving function arguments. Such equalities are often needed to verify enforcement of high-level policies; for example, to verify enforcement of (P1) in Figure 5.2, the analysis must track equalities involving the `id` argument, which identifies the user whose record is being accessed. `callMap` represents all interactions between components, regardless of the actual communication mechanism.

Given a component C and a function F in its API, `callMap`(C, F) returns a set of tuples of the form $(\mathit{calledBy}, R, F', \mathit{args})$, each describing a possible call made during execution of that function. The above tuple represents a call to function F' (the "target function") of the "target" resource (component or IR) R. *calledBy* is analogous to a setuid flag. If *calledBy*=`self`, the target resource sees the user executing the calling component C as the caller; if *calledBy*=`caller`, it sees the user that called F on C as the caller. *args* characterizes the possible arguments of the call to the target function. *args* is represented as a set of equalities of the form $\mathit{attrib} = \mathit{val}$, where *attrib* is an attribute name (recall that we model function arguments as attributes of an operation object), and *val* can be a constant, the name of an attribute (meaning that attribute *attrib* of the target call equals attribute *val* of the enclosing call to F), or `newVar` (meaning that a fresh variable will be used in the analysis to represent this value).

For example, `callMap(solar, getTranscript)` contains the tuple `(self, academicDB,`
`readField, {id=id, field='transcript'})`. The values of `callMap` for `solar`'s `getSSN` and `getCitizenship` functions are similar. `callMap(browser`$_1$`, request)` contains a tuple for every function of every other component, with `newVar` arguments, reflecting that `browser`$_1$ is untrusted and may make arbitrary calls.

When analyzing the security of a design, the `callMap` for each component is based on the component's behavior as described in the design. For an implemented system, `callMap` could be determined from the code. Determining it accurately might be difficult, but an over-approximation can safely be used when verifying enforcement of high-level policies. Over-approximations in `callMap` may cause false alarms, but in many cases, the low-level permit policy of the target component or an intervening component will block the spurious calls or nested calls they make, preventing false alarms. If the analysis does raise false alarms, the corresponding call chains indicate exactly what assumptions about possible calls and their arguments are needed for enforcement of the high-level policies, and the `callMap`, permit policies, or system architecture can be refined accordingly.

5.1.6 Hosts and Firewalls.

Each component has an attribute `type`. This attribute can have any value, but the values `host` and `firewall` have special significance. Hosts and firewalls are hardware components with network connections. Network connectivity is modeled by the relation `link(`C_1`, `C_2`)`, which means that the network may contain a path between C_1 and C_2 that does not pass through a host or firewall. This reflects the fact that we explicitly model hosts and firewalls but not routers. By taking all paths in the network topology into account in the `link` relation, we are making no assumptions about routing (or its security), although such assumptions could be used to restrict the `link` relation.

Hosts, like all components, have attributes, e.g., the set of users with accounts on the host. Since each software component must run on a host, we introduce a relation `runs-on(`C`, `H`)`, which means that component C may run on host H. Hosts provide various services, notably communication services, to components running on them. Host-based security mechanisms may limit the communication performed by a component, e.g., blocking connections with components on untrusted hosts. Firewalls provide a similar security mechanism, typically forwarding some messages and dropping others, based on the firewall's local policy. An obvious way to capture this is to model network security mechanisms as they are implemented (e.g., at the packet level). However, this level of detail would unnecessarily complicate the model and slow the analysis. We adopt a higher-level view, in which hosts and firewalls are modeled as forwarding (or dropping) inter-component function calls, rather than packets. We include relevant network-layer information, such as the source and destination network addresses, as attributes of the operation object Op representing the call. With this approach, the API of a host or firewall includes the operations (of other components) that it forwards; its low-level permit policy allows calls that it forwards and denies calls that it drops; and its `callMap` normally indicates that the call gets forwarded with unchanged arguments.

5.1.7 Low-Level Policies.

Low-level policies for all components are represented in a common rule-based language. The actual configuration languages of the access control mechanisms get translated to this common language; this can be automated. Low-level policy rules have the same form as high-level policy rules. They define auxiliary relations (if desired) and the relation `permit(`U`, `R`, `Op`, `M`)`, where the user U, resource R, and operation Op are the same as for `hPermit`, and M mode M describes the communication mechanism through which the operation is invoked. The mode M enables us to model the fact that different functions may be offered through different interfaces or with different policies. To avoid irrelevant details and distinctions about communication mechanisms, we define modes that reflect how the communication mechanism relates to the system architecture. A mode M has an attribute `type` whose possible values are: `direct`, indicating that the function is called by a user directly executing/running the component; `local`, indicating that the function is called via some inter-process communication mechanism by another component on the same host; or `remote`, indicating that the function is called over the network via some communication mechanism. The mode M may have additional attributes, depending on its type. If M`.type=local`, M`.requester` identifies the calling component. If M`.type=remote`, the attributes M`.srcIP`, M`.srcPort`, M`.destIP`, and M`.destPort` represent the source IP address, source port, destination IP address, and destination port, respectively.

We could express low-level policies in an existing language for attribute-based access control, such as OrBAC [ABB$^+$03], which offers useful abstractions for structuring policies. Our language is simple but flexible and expressive: those abstractions can easily be represented in our language using auxiliary relations, and making them built-in would complicate our analysis algorithm without providing any additional leverage.

Figure 5.3 contains low-level policies for the student information system. `campusIPaddr(`*IPaddr*`)` is an auxiliary predicate that holds if the given IP address is part of the campus network.

5.2 Verification of Enforcement

This section sketches an algorithm for verifying that the low-level policies and system architecture together enforce the high-level policies. For simplicity, the algorithm assumes that the policies does not contain recursion. This restriction is satisfied by most policies and can easily be relaxed if necessary.

```
firewall:
  permit(User, Resource, Op, Mode) <-
    Resource in {webServer, solar}, Mode.type = remote,
    Mode.destPort = 443

solar:
  permit(User, solar, Op, Mode) <-
    Op.function in {getTranscript, getSSN, getCitizenship},
    Op.recordId = User.id, Mode.type = remote
  permit(User, solar, Op, Mode) <-
    User.role = ‘GradSchlClerk’, Op.function = getTranscript,
    Mode.type = remote, campusIPaddr(Mode.srcIP)

webServer:
  permit(_, solar, _, _)

dbServer:
  permit(User, Resource, Op, Mode) <-
    Resource in {academicDB, personalDB}, Mode.type = remote,
    Mode.destPort = 8000

personalDB:
  permit(User, personalDB, Op, Mode) <-
    User.role = ‘Registrar’, Op.function = readRecord,
    Mode.type = remote, campusIPaddr(Mode.srcIP)
  permit(User, personalDB, Op, Mode) <-
    User.role = ‘solar’, Op.function = readField,
    Mode.type = remote
  permit(User, personalDB, Op, Mode) <-
    User.role = ‘admin’, Op.function = addRecord,
    Mode.type = direct

academicDB:
  permit(User, academicDB, Op, Mode) <-
    User.role = ‘solar’, Op.function = readField,
    Mode.type = remote
  permit(User, academicDB, Op, Mode) <-
    User.role = ‘admin’, Op.function = addRecord,
    Mode.type = direct
```

Figure 5.3: Low-level policies for student information system

The default starting points for requests are all functions s_f of all components s_r that can be directly invoked . At each starting point, the arguments to the (top-level) function call and the identity of the user making the call are represented by variables. The algorithm computes all possible chains of functions call that can propagate from each starting point through the system, based on the system architecture and `callMap`. Note that these call chains, ignoring the arguments to each function, correspond to the "context" argument of `hPermit` in the high-level policy. If the call map contains cycles, the number of call chains may be infinite. If a possible call C would extend a call chain with a call that is the same, modulo renaming of variables introduced by `newVar`, as a call already in the call chain, then that call is not explored. To ensure this condition is sound, we include in the policy language only selected functions for accessing the context; currently, we include `head()` and `contains(`*expr*`)` (not, e.g., `length()`).

While constructing call chains, the algorithm accumulates constraints on the values of variables (the starting variables and variables introduced by `newVar`) that represent function arguments; the constraints express that the calls in the chain are permitted by the low-level policies of the components involved (including hosts and firewalls). Values of function arguments obtained from `callMap` are reflected in the formula as equality conjuncts; for example, if `callMap` indicates that a function call represented by `Op1` has `CS` as the value of the `dept` argument, `Op1.dept = CS` is conjoined to the formula. The constraint for a call is determined by matching the conclusions of the `permit` rules in the low-level policy of the component with the call, and, for each rule that matches, instantiating the variables in the rule based on the match and then backchaining to construct a first-order logic formula representing conditions under which the instantiated conclusion can be derived. Since we assume the policy rules are not recursive, the backchaining always terminates. If the accumulated constraint becomes unsatisfiable, the algorithm does not explore extensions of that call chain.

For each call chain S (including prefixes of longer call chains), the algorithm checks whether the call chain is consistent with the high-level policy. Specifically, let Ψ_L be the constraint computed for S, and let C be the context defined by S, i.e., $S[i]$ is a call to function $\mathit{first}(C[i])$ of component $\mathit{second}(C[i])$, where *first* and *second* return the indicated components of a tuple. Call chain S is consistent with the high-level policy if, for every instantiation of the variables that satisfies Ψ_L (in other words, S is feasible), the instantiated call $\mathit{last}(S)$ with context C is permitted by the high-level policy. To check this efficiently, we use backchaining to compute a first-order logic formula Ψ_H representing the conditions (including conditions on the context) under which the call $S[n]$ is permitted by the high-level policy, using a variable V to represent the call's context, and then we check whether the formula $(V = C) \wedge \Psi_L \wedge \neg\Psi_H$ is satisfiable. The satisfiability of this formula implies an inconsistency in the system. Our current prototype uses Yices (`http://yices.csl.sri.com/`) for this purpose. If the satisfiability check succeeds, the logic tool can provide an instantiation of the variables for which the formula is true; this instantiation of S is a counterexample that illustrates how the high-level policy can be violated.

The following example illustrates how our analysis works and how it can identify vulnerabilities. For this example, we modify the low-level policies in Figure 5.3 as follows: the rule for `GradSchlClerk` in `solar`'s low-level policy is removed and replaced with the following rule in the low-level policy for `academicDB`:

```
permit(User, academicDB, Op, Mode) <-
   User.role = ‘GradSchlClerk’, Op.function = readField,
   Op.field = ‘transcript’, Mode.type = remote,
   campusIPaddr(Mode.srcIP)
```

Consider a call chain that propagates along the following path (i.e., context) `C0`: `[(browser`$_2$`, request), (internalHost, request), (dbServer, readField), (academicDB, readField)]`. The constraint associated with S is (note: when it is necessary to rename a variable in a rule during backchaining, in order to avoid name collisions, the algorithm appends the name of the component that the rule is for and/or a sequence number; variables characterizing the top-level call, such as `User` and `Op` in the formula below, never get renamed):

Ψ_L : `Mode_academicDB.type = remote` $\wedge$ `Mode_academicDB.destPort = 8000` $\wedge$ `Op.function = readField` $\wedge$ `Op.field = ‘transcript’` $\wedge$ `User.role = ‘GradSchlClerk’` $\wedge$ `campusIPaddr(Mode_academicDB.srcIP)`

The last call in this chain is to function `readField` of component `academicDB`, which implements `academicIR`. The following constraint is computed for this function call from the high-level policy:

Ψ_H : `Op.function = readField` $\wedge$ `Op.field = ‘transcript’` $\wedge$ `User.role = ‘GradSchlClerk’` $\wedge$ `Context.contains(solar)` $\wedge$ `runs-on(Context.head(), H)` $\wedge$ `internalHost(H)`

The formula (`Context` $=$ `C0`$\wedge\Psi_L)\wedge\neg\Psi_H$ is satisfiable; note that the conjunct `Context.contains(solar)` in Ψ_H is not satisfied when `Context` $=$ `C0`. This shows that the modified low-level policy does not enforce the high-level policy. The significance of this violation depends on why the high-level policy requires that `solar` be in the context for these accesses. For example, `solar` might be responsible for logging accesses to student transcripts by grad school clerks, for compliance with student privacy regulations. Such an error might not be noticed during system execution, while our analysis exposes it during the design stage.

5.3 Trusted Computing Base

In general, a *trusted computing base* (TCB) consists of the hardware and software responsible for enforcing a security policy. We define a set T of components to be a TCB for resource (component or IR) r in system S (a system is defined by sets of components and IRs, with their attributes; `links`, `runs-on`, and `implements` relations; and low-level policies for each component) with high-level policy H if "correct" behavior by the components in T (i.e., behavior consistent with their low-level policy and `callMap`) is sufficient to ensure that all call chains that end at r are consistent with H. Recall that consistency of a call chain with a high-level policy is defined at the end of Section 5.2.

More formally, to check whether T is a TCB for enforcement of the high-level policy for r in system S with high-level policy H, we construct a variant $\texttt{relax}(S, \bar{T})$ of the system, where $\bar{T}$ (the complement of T) is the set of components of S not in T, and then use the method described in Section 5.2 to check whether call chains in that system that end at r are consistent with H. The variant $\texttt{relax}(S, \bar{T})$ is the same as system S except that, for every component C in $\bar{T}$, the low-level permit policy of C is replaced with the single rule `permit(User, Resource, Op, Mode) <- true`, and for every function F in C's API, $\texttt{callMap}(C, F)$ returns the set containing all tuples of the form $(\mathit{calledBy}, R', F', \mathit{args})$ such that $\mathit{calledBy} \in \{\texttt{self}, \texttt{caller}\}$, R' is a component or IR of S other than C, F' is a function in the API of R', and *args* maps all parameters of F' to `newVar`.

Designers might want to specify conditions on the acceptable TCB for a resource—for example, that the TCB for a resource contains only components with specified administrators. Our TCB analysis provides a basis for checking such properties.

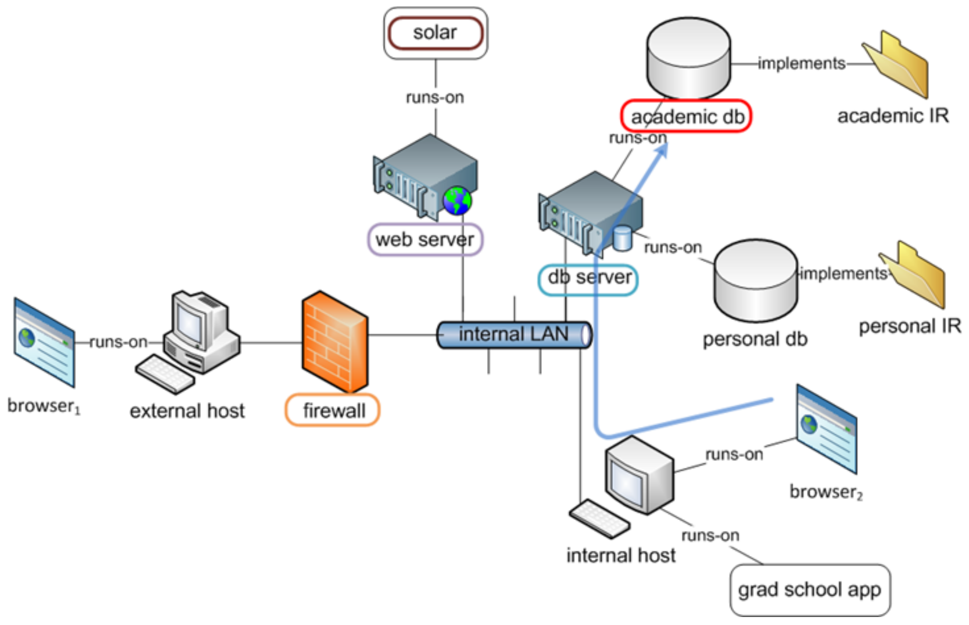


Figure 5.4: The verification algorithm on student information system returns evidence of vulnerability along one path from `browser`$_2$ to `academicDB`.

5.4 Implementation and Evaluation

We implemented a policy development environment based on our framework to evaluate it on case studies based on a university. The environment was implemented in Java with approximately 12000 lines of code. As mentioned earlier, the implementation uses Yices [Yic] which is an SMT solver to test satisfiability of $(\texttt{Context} = \texttt{C0} \wedge \Psi_L) \wedge \neg\Psi_H$ formula. In case the formula is found satisfiable, the instantiations returned by Yices is used to generate the counterexamples to prove system vulnerability.

The implementation was tested on some variants of the student information system described in this chapter. For the example presented in here the program returns a Yices *evidence* which demonstrates the vulnerability. The high-level policy requires that the request context contain `solar` for a `readField` function invocation on `academicDB`. However, as shown in figure 5.4, this is not the case for all possible allowed accesses.

5.5 Related Work

5.5.1 Coordination of Policies in Distributed Systems

Firmato [BMNW99] is a higher-level language for specifying firewall policies. Firmato policies get translated into rule-sets for different models of firewalls, insulating administrators from the details of each model's configuration language. In addition, given the network topology, each firewall's policy can be specialized to contain only the rules relevant to traffic that may pass through it. Work on Firmato does not consider verification of firewall policies against overall network security requirements or analysis of how firewall policies interact with security policies of other components.

García-Alfaro, Cuppens, and Cuppens-Boulahia [GACCB06] define and give algorithms to detect several specific kinds of anomalies (inconsistencies and potential errors) in network security configuration, specifically, configuration of firewalls and network intrusion detection systems (NIDS). In contrast, our work is aimed at verification of general application-level security requirements, taking network security configuration into account but in less detail. Thus, the kinds of properties verified, and the analysis algorithms used, are quite different.

Ioannidis *et al.* [IBI$^+$07] propose the concept of *virtual private services* (VPSs) to describe a service implemented by a collection of components whose security policies must be configured in a coordinated way to enforce an access control policy associated with the service. They express all access control policies in the same language, namely KeyNote [BFK99], without distinguishing "high-level" and "low-level" policies. A policy for a VPS can be delocalized—in particular, its enforcement might involve multiple components—but is otherwise basically a low-level policy, in our terminology. They describe a system architecture for deploying and enforcing policies. They do not consider formal analysis, verification, or refinement of policies.

Cassasa Mont, Baldwin, and Goh [MBG00] developed a general model and tool for template-driven policy refinement. A template captures expert knowledge about how to map policy goals of a specified form to low-level device configuration. Their focus is on tool support for writing and applying such templates. They do not consider automated analysis or verification of templates or policies.

Bandara, Lupu, Moffett, and Russo [BLMR04] propose a formal methodology for policy refinement, based on event calculus [BLR03]. Since most policies today are developed in *ad hoc* ways, not using a formal refinement methodology, we focus instead on verification of given low-level policies against given higher-level policies (requirements). Also, their framework is completely generic; in order to use it for refinement of enterprise security policies, one would need to introduce relations and rules similar to those used in our framework to model system architecture and access control policies.

Sheyner, Haines, Jha, Lippmann, and Wing [SHJ+02] present a method to efficiently construct *attack graphs*, which represent attacks involving sequences of exploits of vulnerabilities in components of a system. Our work is largely complementary to attack graph analysis. Attack graphs are based primarily on vulnerabilities in components; access control policies and calling behavior are not considered, except when they affect a vulnerability. Also, attack graphs are generally used to find violations of system-level security requirements (e.g., who may login to a host), not application-level security policies.

Jürgens developed UMLsec [J0̈5], a security-oriented extension of UML, and, with collaborators, developed various analyses based on it. Among those, the one most closely related to our work is the analysis of network security architecture in [JSB08], which however does not consider the interaction of network security configuration with access control policies of other (non-network) components to achieve application-level security goals. Our analysis could be formulated in terms of UML or an extension of it, but we opted to use a simpler framework for now.

Seehusen and Stølen [SS06] address behavioral refinement for systems with information flow security requirements. Such requirements are not preserved by standard notions of refinement. They define a more discriminating notion of behavioral refinement and show that, under some conditions, information flow properties are preserved by such refinement. Our work differs from their work by considering access control, rather than information flow, by focusing on refinement of access control policies rather than overall system behavior, and by focusing on verification algorithms rather than semantics.

Chapter 6

Conclusions and Future Work

This chapter summarizes the contributions of this thesis and discusses directions for future work.

6.1 Summary of Contributions

6.1.1 Access Control and Administration Using Rules

Administrative Model for Rule-Based Access Control. Many rule-based access control languages have been proposed, but relatively little attention has been paid to administrative models for such languages. The administrative model in [Bec09, BN10] supports addition and removal of facts but not rules. To the best of our knowledge, our ACAR framework is the first that provides fine-grained administrative control of addition and removal of rules as well as facts. This makes ACAR a substantially more powerful administrative access control language. Even though this expressiveness comes at the cost of certain language restrictions, presented in chapter 2, the healthcare network case study in chapter 3 demonstrates that complex and realistic administrative policies can be expressed in ACAR. In the healthcare network case study, a network wide administrative policy allows administrators at various facilities within the network to write policies custom-fit to their individual facilities. Further, these facility administrators can add customized administrative rules that allow specific users within, say, workgroups to act as administrators and add rules and facts that affect their workgroups. This substantially increases the usability of ACAR as an administrative language for large enterprises with decentralized administration.

Algorithm for Abductive Reachability Analysis. We identified and defined the *all-solutions abductive reachability problem* for administrative rule-based access control and presented a sound but incomplete algorithm for it. To the best of our knowledge, this is the first reachability analysis for administrative framework that allows addition and removal of both rules and facts. The algorithm has been implemented in Objective Caml (OCaml). The implementation closely resembles the pseudocode in this thesis. The prototype has been tested on part of the healthcare network case study and runs quite efficiently.

6.1.2 Verification of Policy Enforcement in Enterprise Systems

A Framework for Expressing System Design and High-level Security Goals. We explicitly identified the characteristics of high-level security policies for an enterprise system and presented a framework and language for formally specifying both high-level security goals for an enterprise system along with low-level system design and component policies.

An Algorithm for Verification of Policy Enforcement. We developed an algorithm for verifying that the low-level policies (configurations) and system design enforce given high-level policies. In case the enforcement is not consistent with the high-level policies, the algorithm returns a specific example demonstrating the vulnerability. This can be used to guide modifications to the system design and configuration to achieve the desired high-level security policies.

Trusted Computing Base Computation. We also presented an algorithm that uses our policy enforcement verification algorithm to compute a trusted computing base for a given resource in the system. This helps the system designer identify the components within the system that are critical in enforcement of the high-level security policies for a given resource.

6.2 Future Work

This section describes directions for future work on Access Control and Administration using Rules.

Wildcards in Negative Premises. The current support for wildcards in ACAR is somewhat limited in the sense that wildcards are allowed only in negative premises for predicates for which there are no `removeFact` permission rules in the policy. While our case study demonstrated that this is acceptable for some realistic policies, this limitation is sometimes undesirable. The reason for imposing this restriction is to be able to derive negative premises such as $!p(\ldots)$, with wildcards as arguments, in the analysis algorithm through abduction alone. If we tried to establish such a premise using removals, it is difficult because we cannot precisely determine, in phase 2, what instances of the predicate need to be removed. One approach is to match the negative premise against the positive ones in the current state, and add the matches as new negative premises. This, however, leads to another issue that other matching instances of the predicate p might get added to the state as the algorithm continues to backchain and they would require removals as well. An appropriate extension to the algorithm, then, would be to detect such a situation in phase 3 and handle it by iterating phases 2 and 3 appropriately.

Repeated Administrative Operations. As mentioned in section 4.4.3, our current abductive reachability algorithm cannot generate plans that involve repeated administrative operations, because of the reuse of nodes in tabling in phase 2. However, repetition of an administrative operation is sometimes required. For example, consider a problem instance in which the initial policy contains no facts and contains the rules

```
permit(U, addFact(p(X))) :- true
permit(U, addFact(q(X))) :- p(X)
permit(U, removeFact(p(X))) :- true
permit(U, addFact(r(X))) :- !p(X), q(X)
g(X) :- p(X), q(X), r(X)
```

To reach goal g(a) with an empty residue, administrators must add p(a), add q(a), remove p(a), add r(a), and then add p(a) again, in that order. The current version of the algorithm would re-use p(a) in phase 2 while generating plans to add q(a) and to derive g(a). In phase 3, a call to mightNeedRepeatedOp function would return true causing the algorithm to return and inconclusive result. A direction for future work is to modify the algorithm such that in phase 3, if mightNeedRepeatedOp returns true because a node n is a child of multiple parents, the algorithm replaces n with multiple nodes, one for each parent, to try to satisfy the ordering constraint. The algorithm would then re-run phase 2, to re-compute derivations for these new nodes, and then repeat phase 3, and so on, until no more administrative operations need to be repeated.

Termination. Our analysis algorithm may diverge on some policies. This is expected, because Becker and Nanz's abductive algorithm (which solves a simpler problem) may diverge, and because reachability for ACAR is undecidable. Undecidability of this problem is a corollary of the proof in [SYGR11] that user-permission reachability is undecidable for ARBAC97 extended with parameters, since ARBAC97 policies can be encoded in ACAR in a straightforward way. The algorithm terminates for the case studies we have considered so far, but more work is needed to identify classes of policies for which the algorithm is guaranteed to terminate.

Abductive Analysis Based on State Exploration. The reachability computation at the heart of our abductive analysis algorithm is based on tabling. A discussed above, tabling re-uses derivations, and this causes problems if repeated administrative actions are required. A direction for future work is to explore other techniques for reachability analysis that avoid this problem. One such approach would be to use a state-exploration based algorithm similar to [SYRG07], extended to handle rule-based policies and to perform abductive analysis. It would also be interesting to compare the performance, in terms of memory usage and running time, of tabling-based and state-exploration-based algorithms for abductive reachability analysis.

Trust Management. The current version of the ACAR framework does not explicitly support trust management and trust negotiation. Extending the language to support such features is not too difficult, as mentioned in section 2.7. Extending the abductive atom-reachability analysis algorithm to take credential gathering and trust negotiation into account is another direction for future work. This is a particularly important direction, because the popularity of cloud computing and distributed systems is increasing the need for access control frameworks that can handle decentralization, trust management, and trust negotiation.

Bibliography

[ABB$^+$03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

[Bec05] Moritz Y. Becker. *Cassandra: Flexible Trust Management and its Application to Electronic Health Records*. PhD thesis, University of Cambridge, October 2005.

[Bec09] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. In *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 203–217, 2009.

[BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[BHA11] Glenn Bruns, Michael Huth, and Kumar Avijit. Program synthesis in administration of higher-order permissions. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 41–50. ACM, 2011.

[BLMR04] Arosha K. Bandara, Emil Lupu, Jonathan D. Moffett, and Alessandra Russo. A goal-based approach to policy refinement. In *5th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 229–239, 2004.

[BLR03] Arosha K. Bandara, Emil C. Lupu, and Alessandra Russo. Using event calculus to formalise policy specification and analysis. In *Proc. 4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2003)*, 2003.

[BMD09] Moritz Y. Becker, Jason F. Mackay, and Blair Dillaway. Abductive authorization credential gathering. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 1–8. IEEE Computer Society Press, July 2009.

[BMNW99] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[BN08] Moritz Y. Becker and Sebastian Nanz. The role of abduction in declarative authorization policies. In *Proc. 10th International Symposium on Practical Aspects of Declarative Languages (PADL 2008)*, pages 84–99. Springer-Verlag, 2008.

[BN10] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. *ACM Transactions on Information and System Security*, 13(3), 2010.

[CLM+09] Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, and Arosha Bandara. Expressive policy analysis with enhanced system dynamicity. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 239–250. ACM, 2009.

[CM86] Eugene Charniak and Drew McDermott. *Introduction to artificial intelligence.* Addison-Wesley series in computer science. Addison-Wesley, 1986.

[CW89] Weidong Chen and David Scott Warren. Abductive resoning with structured data. In *NACLP*, pages 851–867, 1989.

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

[GACCB06] Joaquín García-Alfaro, Frédéric Cuppens, and Nora Cuppens-Boulahia. Analysis of policy anomalies on distributed network security setups. In *Proc. 11th European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *Lecture Notes in Computer Science*, pages 496–511. Springer, September 2006.

[GS09] Puneet Gupta and Scott D. Stoller. Verification of security policy enforcement in enterprise systems. In *Proceedings of the 24th IFIP International Information Security Conference (SEC)*, volume 297 of *IFIP Advances in Information and Communication Technology*. Springer-Verlag, May 2009.

[GSX11] Puneet Gupta, Scott D. Stoller, and Zhongyuan Xu. Abductive analysis of administrative policies in rule-based access control. In *Proc. Seventh International Conference on Information Systems Security (ICISS 2011)*, volume 7093 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, December 2011.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[IBI$^+$07] Sotiris Ioannidis, Steven M. Bellovin, John Ioannidis, Angelos D. Keromytis, Kostas G. Anagnostakis, and Jonathan M. Smith. Virtual private services: Coordinated policy enforcement for distributed applications. *International Journal of Network Security*, 4(1):69–80, January 2007.

[JÖ5] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[JLT$^+$08] Somesh Jha, Ninghui Li, Mahesh Tripunitara, Qihua Wang, and William Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.

[JSB08] Jan Jürjens, Jörg Schreck, and Peter Bartmann. Model-based security analysis for mobile communications. In *30th International Conference on Software Engineering (ICSE)*, pages 683–692, 2008.

[KKT92] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[KN07] E. Kleiner and T. Newcomb. On the decidability of the safety problem for access control policies. *Electr. Notes Theor. Comput. Sci.*, 185:107–120, 2007.

[LM07] Ninghui Li and Ziqing Mao. Administration in role based access control. In *Proc. ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, pages 127–138. ACM Press, March 2007.

[LT06] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and System Security*, 9(4):391–420, November 2006.

[MBG00] Marco Casassa Mont, Adrian Baldwin, and C. Goh. POWER prototype: Towards integrated policy-based management. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2000. Also appeared as Tech Report HPL-1999-126, HP Labs Bristol, 1999.

[Nat03] National Health Service of the United Kingdom. Output based specification for integrated care record service version 2, August 2003. Available via `http://www.dh.gov.uk/`.

[Pea87] Judea Pearl. Embracing causality in formal reasoning. In *AAAI*, pages 369–373, 1987.

[RND77] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, February 1999.

[SCFY96a] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SCFY96b] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SHJ$^+$02] Oleg Sheyner, Joshua W. Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284, 2002.

[SS06] Fredrik Seehusen and Ketil Stølen. Maintaining information flow security under refinement and transformation. In *Proc. 4th International Workshop on Formal Aspects in Security and Trust (FAST)*, volume 4691 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 2006.

[SYGR11] Scott D. Stoller, Ping Yang, Mikhail Gofman, and C. R. Ramakrishnan. Symbolic reachability analysis for parameterized administrative role based access control. *Computers & Security*, 30(2-3):148–164, March-May 2011.

[SYRG07] Scott D. Stoller, Ping Yang, C. R. Ramakrishnan, and Mikhail 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.

[SYSR11] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C. R. Ramakrishnan. Policy analysis for administrative role based access control. *Theoretical Computer Science*, 2011.

[The] The Office of the National Coordinator for Health Information Technology. Nationwide health information network. `http://healthit.hhs.gov/`.

[Uni03] United States Department of Health and Human Services. Summary of the HIPAA privacy rule, May 2003. `http://www.hhs.gov/ocr/privacy/hipaa/understanding/summary/index.html`.

[XSB] XSB. Available at http://xsb.sourceforge.net/.

[Yic] Yices: An SMT Solver. Available at http://yices.csl.sri.com/.