# Specifying and Reasoning about Dynamic Access-Control Policies

Daniel J. Dougherty,[1] Kathi Fisler,[1] and Shriram Krishnamurthi[2]

[1] Department of Computer Science, WPI
[2] Computer Science Department, Brown University
dd@cs.wpi.edu, kfisler@cs.wpi.edu, sk@cs.brown.edu

**Abstract.** Access-control policies have grown from simple matrices to non-trivial specifications written in sophisticated languages. The increasing complexity of these policies demands correspondingly strong automated reasoning techniques for understanding and debugging them. The need for these techniques is even more pressing given the rich and dynamic nature of the environments in which these policies evaluate. We define a framework to represent the behavior of access-control policies in a dynamic environment. We then specify several interesting, decidable analyses using first-order temporal logic. Our work illustrates the subtle interplay between logical and state-based methods, particularly in the presence of three-valued policies. We also define a notion of policy equivalence that is especially useful for modular reasoning.

## 1 Introduction

Access control is an important component of system security. Access-control policies capture rules that govern access to data or program operations. In the classical framework [28], a policy maps each user, resource and action to a decision. The policy is then consulted whenever a particular user wants to perform an action on a resource. The information that defines this user, resource, and action forms an access *request*.

Modern applications increasingly express policies in domain-specific languages, such as the industrially popular language XACML [33], and consult them through a policy-enforcement engine. Separating the policy from the program in this manner has several important consequences: it allows the same policy to be used with multiple applications, it enables non-programmers to develop and maintain policies, and it fosters rich mechanisms for combining policy modules [9, 33] derived from different, even geographically distributed, entities. (In XACML, a typical combiner is "a decision by one module to deny overrides decisions by all other modules".) A university administration can, for example, author a common policy for campus building ID-card locks; each department can individually author a policy covering its unique situations (such as after-hours access for undergraduate research assistants); and an appropriate policy combiner can mediate the decisions of the two sub-policies.

Access-control policies are hard to get right. Our appreciation for the difficulty of authoring policies stems from our experience maintaining and debugging the policies from a highly-configurable commercial conference paper manager called CONTINUE [27]. Almost all interesting bugs in CONTINUE have related to access control in some form.

1. During the submission phase, an author may submit a paper
2. During the review phase, reviewer $r$ may submit a review for paper $p$ if $r$ is assigned to review $p$
3. During the meeting phase, reviewer $r$ can read the scores for paper $p$ if $r$ has submitted a review for $p$
4. Authors may never read scores

1. During the submission phase, an author may submit a paper
2. During the review phase, reviewer $r$ may submit a review for paper $p$ if $r$ is not conflicted with $p$
3. During the meeting phase, reviewer $r$ can read the scores for paper $p$ if $r$ has submitted a review for $p$ and $r$ is not conflicted with $p$
4. Authors may never read scores

**Fig. 1.** Two candidate policies for controlling access to review scores.

Many sources of complexity make policies difficult to author. Combiners are one natural cause of difficulty. Size is another factor: policies in realistic applications can govern hundreds of actions, resources, and classes of users (called roles). Perhaps most significantly, decisions depend on more than just the information in the access request. Consider the policy governing PC member access to conference paper reviews: a reviewer assigned to a paper may be required to submit his own review before being allowed to read those of others. The conference manager software maintains the information about which reviewers have submitted reviews for which papers; the policy engine must be able to consult that information when responding to an access request. Such information forms the *environment* of the policy. As this simple example shows, environment data may be highly dynamic and affected by user actions.

What is the impact of the environment? Figure 1 shows two candidate policies governing access to review scores for papers in a conference manager. Which policy should we choose? The policies differ syntactically only in rules 2 and 3 but, if the application allows conflict-of-interest to change after paper assignment, the *semantic* change is considerable. Imagine a reviewer who is initially assigned a paper and submits a review, but the PC chair later learns that the reviewer was conflicted with the paper. By the policy on the left, the reviewer can read the scores for the conflicted paper.

As the example shows, such leaks are not evident from the policy document alone: they require consideration of the dynamic environment. Fixing these, however, requires edits to the *policy*, not the program. This suggests that analysis should focus on the policy, but treat information from the program as part of the policy's environment.

Whereas existing work on reasoning about access-control policies models the environment only lightly, if at all, this paper presents formal analyses for access-control policies in their dynamic environments. We propose a new mathematical model of policies, their environments, and the interactions between them. We then propose analyses that handle many common scenarios, focusing on two core problems: *goal reachability* and *contextual policy containment*. Such analyses require a combination of relational reasoning (to handle interesting policies) and temporal reasoning (for the environments). In addition, the analyses must support realistic development scenarios for policies, such as modular policy authoring and upgrading. A recurring theme in this

Permit($a$, submit-paper, $p$) :– author($a$) , paper($p$) , phase(submission)
Permit($r$, submit-review, $p$) :– reviewer($r$) , paper($p$), assigned($r$, $p$) , phase(review)
Permit($r$, read-scores, $p$) :– reviewer($r$) , paper($p$), has-reviewed($r$,$p$) , phase(meeting)
Deny($a$, read-scores, $p$) :– author($a$) , paper($p$)

**Fig. 2.** Formal model of policy on left in Figure 1.

work is the interplay between techniques for defining these analyses originating from formal verification and from databases.

## 2 Modeling Policies and their Dynamic Environments

The sample policies in Figure 1 require information such as the assignment of papers to reviewers and conflicts of interest between reviewers and papers. Policies are declarative statements over data from requests and over relations that capture information gathered by the application (such as conflict-of-interest data). Following many other policy models [5, 12, 23, 29, 30], we capture policies as Datalog programs.

A Datalog rule is an expression of the form

$$R_0(\vec{u}_0) \text{ :– } R_1(\vec{u}_1), \ldots, R_n(\vec{u}_n)$$

where the $R_i$ are relation names, or *predicates*, and the $\vec{u}_i$ are (possibly empty) tuples of variables and constants. The *head* of the rule is $R_0(\vec{u}_0)$, and the sequence of formulas on the right hand side is the *body* of the rule. Given a set of Datalog rules, a predicate occurring only in the bodies of rules is called *extensional* and a predicate occurring in the head of some rule is called *intentional*. For a set of rules $P$, $edb(P)$ and $idb(P)$ denote the extensional and intentional predicates of $P$, respectively. A policy is *recursive* if some idb appears in a rule body. The *signature* of $P$, $\Sigma_P$, is $edb(P) \cup idb(P)$. A set of *facts* is a set of closed atomic formulas over a signature $\Sigma$.

**Definition 1.** Let Subjects, Actions, and Resources each be sorts. Let $\Sigma$ be a first-order relational signature including at least the two distinguished ternary predicates Permit and Deny of type Subjects $\times$ Actions $\times$ Resources.[3] A *policy rule* over $\Sigma$ is a Datalog rule over $\Sigma$ whose head is either Permit or Deny. A *policy* over $\Sigma$ is a set of policy rules over $\Sigma$.

That is, a policy is a set of Datalog rules whose idb predicates are amongst Permit and Deny. We use an explicit Deny relation following the XACML policy language [33], rather than interpret deny as the negation of permit, to allow a policy to not apply to some requests. The distinction between denial and non-applicability is useful for decomposing policies into sub-policies that only cover pertinent requests, as in the university example of the Introduction. (Bertino *et al*. discuss implications of supporting

[3] Subjects, Actions, and Resources could have more structure, such as tuples to model resources with attributes or sets of Subjects to model joint actions. Such changes do not affect our theoretical foundations, so we use the atomic versions to simplify the presentation.

negated decisions [8].) We point out the consequences of this decision on our models and analyses as they arise in the paper.

Figure 2 shows a sample policy. The policy governs the use of the actions submit-paper, submit-review, and read-scores based on information from the environment.

What is an environment? A principal source of environment information is the program (e.g., which reviewers have submitted papers). Some information comes from end-users (such as credentials). The run-time system also provides information (such as the current time), and some information comes from the policy framework itself (in role-based access control, for example, policies operate under assignments of users to roles and under hierarchies of permission inheritance among roles). These diverse sources suggest that (i) the environment must be a transition system, to model the program's execution and the passage of time, and (ii) each state must consist of an instance of the edb relations referred to by the policy. This model is therefore in the family of recent work on representing programs as transitions over relations [2, 13, 41, 44]. Because our model is general enough to handle most forms of environment information, we focus on the general model and ignore finer distinctions in the rest of this paper.

Concretely, consider the policy in Figure 2. The predicate has-reviewed tracks which reviewers have submitted reviews for which papers. When a reviewer $r$ submits a review for paper $p$, the tuple $\langle r, p\rangle$ is added to has-reviewed in the policy environment. The phase predicate tracks the current phase of the reviewing process. When the PC chair ends the review phase and starts the program committee meeting, the fact phase(review) is removed from the set of current facts and phase(meeting) is added.

Semantically, at any given time the set **E** of facts in the environment relevant to the policy rules constitute an instance over the edb relations of $P$. Evaluation of access requests, such as Permit$(s,a,r)$, can thus be viewed as asking for the truth of the sentence Permit$(s,a,r)$ in this structure. More constructively, it is well-known that a set $P$ of Datalog rules defines a monotone operator on the instances over $\Sigma_P$. In this vein, $P$ inductively defines instances of idb names in terms of **E**, as follows. Treat **E** as an instance over $\Sigma_P$ by adding empty relations for the idb names, and take the least fixed-point of the operator determined by $P$ starting with **E**. The idb relations in the resulting instance are the defined relations. We call the generated idb facts the *access tables* of $P$ with respect to **E** (denoted $P(\mathbf{E})$). Negation can be introduced into the framework with some conceptual and computational cost [1].

Transitions in the policy's environment are triggered by various conditions. Some arise from the passage of time (such as the passing of the submission deadline moving the conference into the review phase). Others arise from user or program actions (once an author submits a review, for example, he can read other reviews for the same paper). We use the generic term *event* for all of these conditions, and assume a signature of events that can label transitions in the environment:

**Definition 2.** Given an event signature $\Sigma_{EV}$, an *event* is a closed instance of one predicate or constant in $\Sigma_{EV}$. An *environment model* over a signature $\Sigma$ relative to event signature $\Sigma_{EV}$ is a state machine $\mathcal{V}$ whose states are relational structures over $\Sigma$ and whose transitions are labeled with events from $\Sigma_{EV}$.

A policy interacts with its dynamic environment by consulting facts in the environment and potentially constraining certain actions in the environment. The latter captures the

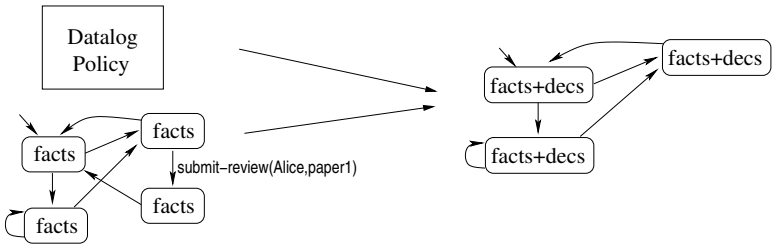


**Fig. 3.** Combining an environment model and a policy into a dynamic access model.

influence of policy decisions on an application that uses it (recall that the policy's environment includes the model of the application). We model such interactions through events that share the same names as actions in policy requests. For example, a transition labeled submit-review(Alice,paper1) would correspond to a request sent to the policy. Not all events need to be governed by the policy. To avoid ambiguity, we require that all predicates that appear in both $\Sigma_{EV}$ and the Actions sort of the policy have the type Subjects $\times$ Resources in $\Sigma_{EV}$.

A policy and an environment model for the policy's dynamic environment combine to form a state machine over access tables, as shown in figure 3 (where "decs" is short for "decisions"). Intuitively, the access tables arise from applying the policy to the facts at each state of the environment model. The transitions are a subset of those in the environment model. Transitions whose event labels are policy actions are kept if the request defined by that event yields Permit in the source state of the transition, and removed if the request yields Deny. Some transitions may be labeled with policy actions for which their source state yields neither Permit nor Deny. Applications must determine whether to permit or deny such actions. Rather than fix an interpretation, we assume that an application specifies which transitions should be treated as denied in the absence of a policy decision. This expectation is reasonable because an application queries the policy engine for decisions and acts on the responses. We use the term *policy context* for a pair containing an environment model and a subset of its transitions, denoted $C = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{E}$ is the set of transitions that the application treats as defaulting to deny.

**Definition 3.** Let $P$ be a policy, $\mathcal{V}$ be an environment model over $\Sigma_P$ and $C = \langle \mathcal{V}, \mathcal{E} \rangle$ be a policy context. The *dynamic access model* for $P$, $\mathcal{V}$, and $\mathcal{E}$ is the state machine $\mathcal{D}$ obtained by

- augmenting each state $q$ of $\mathcal{V}$ with the access tables from evaluating $P$ at $q$; then
- eliminating transitions $t$ that are labeled with policy actions such that (i) $P$ does not yield Permit, and (ii) either $P$ yields Deny or $t$ is in $\mathcal{E}$, and
- eliminating unreachable states.

We use the term $(C,P)$-*accessible* for states in $\mathcal{D}$. We say "accessible" rather than "reachable" to connote the influence of the policy: states in $\mathcal{D}$ are reached only by securing the permission of the access-control policy.

Dynamic access models satisfy the definition of environment models. This allows incremental construction of dynamic access models from a series of policy modules. The definition assumes that the policy will not yield both Permit and Deny for any request considered in the second clause. If this assumption is violated, we say the policy has no model. The subsequent results in this paper assume that policies have models.

The following remark will be useful later.

*Remark 4.* Let $C = \langle \mathcal{V}, \emptyset \rangle$ be a policy context with an empty set of default-deny transitions and let $P$ and $P'$ be policies. Then any state which is $(C, P \cup P')$-accessible is $(C, P)$-accessible.

Ideally, environment models would be at least partially derived from applications. Standard techniques such as abstract interpretation [11] address this problem. Such techniques are commonly used in software verification, and are not discussed further in this paper. In general, we expect that finite models over-approximate their original infinite models, so that all sets of facts reachable in the original model remain reachable in the abstracted model.

## 3 Analyzing and Comparing Policies

Formal analyses can answer many useful questions about policies. Two fundamental analyses are *safety* (does a policy prohibit users from doing something undesirable) and *availability* (does a policy permit a user to do something that they are allowed to do). Both of these depend on the dynamic environment and resemble properties common to model checking.

Policy authors also need the ability to compare policies in the absence of formal properties. Policies require upgrades and revisions just as programs do. Authors need to know that their policies implement expected changes, but more importantly, that the change did not yield *unanticipated* changes to decisions. Property-based verification is of limited use for this problem as it would require the policy author to write properties expressing the unanticipated changes. Analyses that compare policies and provide insights into the requests on which they yield different decisions are therefore crucial. This section formalizes analyses both on single-policies and for comparing policies.

### 3.1 Goal Reachability

The analyses for safety and availability (a form of liveness) share a similar structure: they ask whether there is some accessible state in the dynamic access model which satisfies some boolean expression over policy facts. Checking whether a policy allows authors to read review scores, for example, amounts to finding an accessible state satisfying the formula

$$\exists x_1 x_2.(\mathsf{Permit}(x_1, \text{read-scores}, x_2) \wedge \mathsf{author}(x_1) \wedge \mathsf{paper}(x_2)).$$

We use the term *goal reachability* for this common analysis problem, where a goal is formally defined as follows:

**Definition 5.** An *n-ary goal* is a sentence of the form $\exists x_1 \ldots x_n \,.\, A$, where $A$ is a Boolean combination of atomic formulas over $\Sigma_P$. A goal is *conjunctive* if $A$ is a conjunction of edbs. A goal is $(C,P)$-*reachable* if it is satisfied in a $(C,P)$-accessible state.

The formulas that capture goals do not interleave quantifiers and temporal operators. When formulas do interleave these, the logic gets complicated if the domains of the structures at different states are allowed to vary (this phenomenon is familiar from predicate modal logic). For problems that require such formulas, FO-LTL is a sublanguage of linear predicate temporal logic that avoids the difficulty with varying-domain models, yet is rich enough to express many properties of interest [13, 41].

Goal reachability combines database query evaluation and reachability analysis. The body of a goal is precisely a database query: to evaluate the goal at a particular state in a model is to evaluate the associated Boolean query on the database of facts at that state. Model checking algorithms for first-order temporal logics subsume this problem [13, 41]. Given that goal reachability is a very useful and special case of first-order model checking, however, it is worth understanding the complexity of goal checking. Although checking the truth of an arbitrary first-order sentence in a finite model is PSPACE-complete, the result of any fixed Datalog query can be computed in polynomial time in the size of the database, and the result of any fixed *conjunctive* query over a database $\mathbf{Q}$ can be computed in space $O(\log|\mathbf{Q}|)$ [42]. Strategies for efficient evaluation of Datalog queries have been much-studied [1], particularly in the case of of conjunctive queries, resulting in many fast evaluation techniques [16].

The following theorem records an upper bound on the asymptotic complexity of deciding goal reachability in models with fixed domains.

**Theorem 6.** *Let $\mathcal{D}$ be a finite dynamic access model with n states, each of which has the same finite domain of size d. Reachability in $\mathcal{D}$ of a fixed goal G can be checked in time polynomial in n and d. If G is a conjunctive goal then reachability can be checked in nondeterministic logspace in the size of $\mathcal{D}$.*

*Proof.* Each state $q$ of $\mathcal{D}$ can be considered as a database $\mathbf{Q}$ over the schema given by the signature of the policy. For a fixed signature the size of $\mathbf{Q}$ is bounded by a polynomial in $d$. Hence the satisfiability of a goal formula at a given state can be computed in polynomial time in $d$. We then use the fact that reachability between nodes in a directed graph is in NLOGSPACE [24] and so requires time polynomial in $n$. When the goal is conjunctive, we require NLOGSPACE to check satisfiability of a goal formula at a given state. Hence, the entire goal reachability test can be done in NLOGSPACE. □

### 3.2 Contextual Policy Containment

Intuitively, policy containment asks whether one policy is more permissive than another. For example, we could say that policy $P_2$ subsumes the decisions of policy $P_1$ if every permitted request under $P_1$ is permitted under $P_2$ and every denied request under $P_2$ is denied under $P_1$. Whether a request is permitted or denied in a policy, though, depends on the set of facts that might support the request. We can exploit our environment model to restrict attention to those sets of facts that are accessible in the environment. This gives rise to the following formal definition of contextual policy containment:

**Definition 7.** Let $P(\mathbf{Q})(\mathsf{Permit})$ denote the Permit table defined by policy $P$ over a set of facts $\mathbf{Q}$ (and similarly for $P(\mathbf{Q})(\mathsf{Deny})$). $P_2$ *contains* $P_1$ *in context* $C$, written $P_1 \preceq^C P_2$, if for all instances $\mathbf{Q}$ of edb and idb facts in $(C, P_1)$-accessible states,

$$P_1(\mathbf{Q})(\mathsf{Permit}) \subseteq P_2(\mathbf{Q})(\mathsf{Permit}) \quad \text{and} \quad P_2(\mathbf{Q})(\mathsf{Deny}) \subseteq P_1(\mathbf{Q})(\mathsf{Deny}).$$

$P_1$ and $P_2$ are *contextually equivalent* if $P_1 \preceq^C P_2$ and $P_2 \preceq^C P_1$.

A subtlety in comparing the semantics of two polices arises due to the fact that changing policies can result in a change in the accessibility relation in a dynamic access model: which states should we examine? The following lemma justifies the choice made in Definition 7.

**Lemma 8.** *If $P_1 \preceq^C P_2$, then every $(C, P_1)$-accessible state is $(C, P_2)$-accessible.*

*Proof.* We induct over the length of a shortest path from the start state to a given $(C, P_1)$-accessible state. It suffices to show that at any such state the set of actions enabled under $P_1$ is a subset of those enabled under $P_2$. But this is clear from an examination of Definition 3. □

Ideally, we would like to use contextual containment to reason about relationships between *fragments* of policies, as well as entire policies. Reasoning about the relationships between policy fragments is critical for policies authored across multiple entities (as in our university example in the Introduction). Modular policy reasoning is subtle, however, in the presence of requests to which the policy does not apply. In our model, the context determines how such requests are handled. If a new policy fragment permits a request that defaulted to Deny in the context, new states could become accessible; these states would have not been tested for containment, thus rendering policy reasoning unsound.

Modular reasoning is sound, however, if policy combination cannot make additional states accessible. If the containment check between two fragments occurs in a context in which all non-applicables default to Permit, for example, policy containment and accessibility lift to modular policy reasoning, as we now show.

**Lemma 9.** *Let $C = \langle \mathcal{V}, \emptyset \rangle$ be a policy context with an empty set of default-deny transitions. If $P_1 \preceq^C P_2$ then for all policies $P$, $(P \cup P_1) \preceq^C (P \cup P_2)$.*

*Proof.* Let $q$ be a state which is $(C, P \cup P_1)$-accessible and let $\mathbf{Q}$ be the associated database of edb and idb facts at $q$. By Remark 4, $q$ is $(C, P_1)$-accessible, so the inclusions in the definition of $P_1 \preceq^C P_2$ apply at $\mathbf{Q}$. Letting $T$ denote the operator constructing the idb relations for a Datalog program, note that the fixed point of $T_{P \cup P_i}$ is the same as that of $T_{P_i} \circ T_P$. By hypothesis, at each iteration $n$ of the fixpoint construction starting with $\mathbf{Q}$, we have $(T_{P_1} \circ T_P)^n(\mathbf{Q}) \subseteq (T_{P_2} \circ T_P)^n(\mathbf{Q})$. The lemma follows. □

Contextual containment under an empty set of default Deny transitions is analogous to uniform containment as defined for Datalog programs [10, 35] (which is itself a generalization of the standard homomorphism-based characterization of containment

for conjunctive queries). Correspondingly, we use the term *uniform contextual containment* for this scenario. Such preservation under context is also the key feature of observational equivalence in programming language theory. For the same reasons that observational equivalence is the canonical notion of equality between programming language expressions, we feel that uniform contextual equivalence should be viewed as a fundamental notion of policy equivalence.

Uniform contextual containment supports the following analyses, none of which require a policy author to write formal properties:

- A new policy $P$ neither removes any permissions nor adds any denials if and only if the new *fragment* of $P$ uniformly contextually contains the fragment it replaced. If the replaced fragment also contains the new fragment, the two policies yield precisely the same decisions.
- A new policy $P'$ adds a specific set of permissions $I$ to an old policy $P$ if $P \cup I$ (where $I$ is a set of idb facts) is uniformly contextually equivalent to $P'$.

Lemma 9 over-approximates the set of accessible states by setting $\mathcal{E}$ to $\emptyset$. Such over-approximation is inherent to an open system setting, where we cannot make assumptions about the behavior of other modules. Naturally, this can result in irrelevant failures to prove containment. This effect can be mitigated: the degree of over-approximation is controlled entirely by the value of $\mathcal{E}$, which is a parameter to the containment check.

### Checking Contextual Policy Containment

We now discuss how to implement a test for contextual policy containment. The most straightforward approach is to rename the predicates in the two policies so they are disjoint (we use subscripts in the formula below), take the union of the two policies, and use model checking to verify the temporal logic sentence

$$\mathsf{AG}\ \forall x_1 x_2 x_3\ .\ ((\mathsf{Permit}_1(x_1,x_2,x_3) \rightarrow \mathsf{Permit}_2(x_1,x_2,x_3)) \\ \wedge (\mathsf{Deny}_2(x_1,x_2,x_3) \rightarrow \mathsf{Deny}_1(x_1,x_2,x_3))).$$

The universal quantification over requests makes this approach potentially expensive to evaluate at each state of the dynamic access model.

We can improve the situation by focusing on the relationship between policies and single rules. Roughly speaking we will reduce the policy containment question to consideration of whether individual rules are contained in (whole) policies. It is natural to consider a single rule as a policy in its own right. But the notion of accessibility is different depending on whether a rule is considered in isolation or as part of a larger policy. We will thus want to explore containment between a rule $\rho$ from policy $P_1$ and a whole policy $P_2$ but restricting attention to states accessible under all of policy $P_1$. This motivates the following refinement of contextual containment.

**Definition 10.** Let $P_1$ and $P_2$ be policies and let $\rho$ be a rule. Then $\rho \preceq^{C}_{P_1} P_2$ if for all instances $\mathbf{Q}$ of edb and idb facts in $(C, P_1)$-accessible states, $\rho(\mathbf{Q})(R) \subseteq P_2(\mathbf{Q})(R)$, where $R$ is the predicate at the head of $\rho$.

Analyzing contextual containment in terms of individual rules will be sound assuming a rather natural constraint on rules: that no Permit rule has the Deny predicate occurring in its body and no Deny rule has the Permit predicate occurring in its body. We will call such polices *separated*. This restriction is naturally satisfied in most policies. Furthermore, note that if only one kind of violation occurs, for example if some Permit rules depend on Deny but not vice versa, then the policy can be rewritten to be separated simply by expanding the offending occurrences of Deny by their definitions.

**Lemma 11.** *Let $P_1$ and $P_2$ be separated policies and let $C$ be a policy context. Then $P_1 \preceq^C P_2$ if and only if for each* Permit *rule $\rho_1$ of $P_1$, $\rho_1 \preceq^C_{P_1} P_2$, and for each* Deny *rule $\rho_2$ of $P_2$, $\rho_2 \preceq^C_{P_1} P_1$.*

*Proof.* Suppose $P_1 \preceq^C P_2$, and let $\rho_1$ be a Permit-rule of $P_1$. Since $P_1$ is separated, at any state $q$ with associated database $\mathbf{Q}$, $\rho_1(\mathbf{Q})(\mathsf{Permit}) \subseteq P_1(\mathbf{Q})(\mathsf{Permit})$; it follows that $\rho_1 \preceq^C_{P_1} P_2$. A similar argument shows that $\rho_2 \preceq^C_{P_2} P_1$ for each Deny-rule $\rho_2$ from $P_2$.

For the converse we consider without loss of generality a fact $\mathsf{Permit}(\vec{u})$ in $P_1(\mathbf{Q})$ for $\mathbf{Q}$ associated with a $(C, P_1)$-accessible state $q$ and argue that $\mathsf{Permit}(\vec{u})$ is in $P_2(\mathbf{Q})$ by induction on the number of stages in the Datalog computation of this fact under $P_1$. Since $P_1$ is separated this computation relies only on edb facts from $q$ and Permit-facts generated in fewer steps by $P_1$, so the result follows. □

We now focus on the problem of testing containments of the form $\rho_1 \preceq^C_{P_1} P_2$ (from Definition 10). While it is tempting to treat this as a purely logical problem, this is insufficient because it might miss relationships among the edb relations being maintained in the dynamic access model. Consider an example in which a policy author wants to replace the following rule $\rho_1$ for reviewers' access to paper reviews with rule $\rho_2$:

$\rho_1$: Permit($r$, read-scores, $p$) :– reviewer($r$) , has-reviewed($r$,$p$) , phase(meeting)
$\rho_2$: Permit($r$, read-scores, $p$) :– reviewer($r$) , assigned($r$,$p$) , phase(meeting)

Suppose the the dynamic access model maintains an invariant that reviews have only been submitted by reviewers who were assigned to a paper. Then $\rho_1 \preceq^C \rho_2$ as single-rule policies since at every state, has-reviewed($r$,$p$) implies assigned($r$,$p$).

A related semantic phenomenon is the following. If every Subject (for example) in a model's domain were named by a constant, it could happen that the effect of a given rule was subsumed by finitely many rules of a policy in a "non-uniform" way.

Such examples illustrate why syntactic analysis is in general insufficient for checking contextual containment. The following algorithm works directly with the policy context $C$ to check containment.

**Algorithm 12** Let $C$ be a policy context, $P_1$ and $P_2$ be policies, and $\rho \equiv R_0(\vec{u}_0)$ :– $R_1(\vec{u}_1), \ldots, R_n(\vec{u}_n)$ be a rule from $P_1$. To test whether $\rho \preceq^C_{P_1} P_2$:

For each $(C, P_1)$-accessible state $q$ and for each valuation $\eta$ mapping the variables of $\rho$ into $q$ which makes each $R_i(\eta\vec{u}_i)$ true, let $\mathbf{Q}^*$ be the database whose edb facts are those of $q$ and whose idb facts are those $R_i(\eta\vec{u}_i)$ where $R_i$ is an idb predicate from $\rho$. If $\eta\vec{u_0} \in P_2(\mathbf{Q}^*)(R_0)$ for each such $\mathbf{Q}^*$, return success; if this fails for some $\mathbf{Q}^*$, fail.

**Lemma 13.** *Algorithm 12 is sound and complete for testing* $\rho \preceq^{C}_{P_1} P_2$.

*Proof.* Suppose $\rho \preceq^{C}_{P_1} P_2$ holds. Let $q$ be a $(C, P_1)$-accessible state, let $\mathbf{Q}$ be the associated database instance and suppose $R(\vec{d})$ is in $\rho(\mathbf{Q})$, where $R$ is the predicate at the head of $\rho$; we want to show that $R(\vec{d})$ is in $P_2(\mathbf{Q})$. The fact that $R(\vec{d})$ is in $\rho(\mathbf{Q})$ is witnessed by an instantiation of the body of $\rho$ with elements from $q$ that comprise edb facts from $q$ and idb facts derived from evaluating $\rho$ as a policy over $q$. But those latter idb facts are part of the instance $\mathbf{Q}^*$ as constructed in Algorithm 12, as are the edb facts from $q$. So the algorithm will report success. The argument that the algorithm correctly reports failures is similar. □

The effect of a rule will often be captured by a policy in the sense of logical entailment, without appealing to the semantics of the application in question. Such relationships can be uncovered by purely symbolic computation: this is essentially the notion of uniform containment between Datalog programs. In our setting this takes the following form. First note that a collection $B$ of atomic formulas can be considered as a database of facts by viewing the variables as values and the formulas as defining tables.

**Definition 14.** Let $P$ be a policy, $\rho \equiv R_0(u_0) \text{ :- } R_1(u_1), \ldots, R_n(u_n)$ be a rule, and $B$ be the database instance derived from the body of $\rho$. *$P$ simulates $\rho$* if $u_0 \in P(B)(R_0)$.

**Lemma 15.** *Let $C$ be a policy context, let $P_1$ and $P_2$ be policies, and let $\rho_1$ be a rule from $P_1$. If $P_2$ simulates rule $\rho_1$ then $\rho_1 \preceq^{C}_{P_1} P$.*

*Proof.* It is easy to see that when $P_2$ simulates $\rho_1$, the computation in Algorithm 12 will succeed at any state $q$. □

Checking rule simulation requires time polynomial in $\rho$ when the schema $\Sigma$ is considered fixed: the complexity is $O(d^k)$ where $k$ is the maximum arity of a predicate in $\Sigma$ and $d$ is the number of distinct variables in $B$.

The following algorithm summarizes how we can combine rule simulation and direct checking of a policy context to test contextual containment.

**Algorithm 16 (Improved containment checking)** Let $C$ be a policy context and let $P_1$ and $P_2$ be separated policies. To test whether $P_1 \preceq^{C} P_2$:

(i) Consider each Permit rule $\rho_1$ of $P_1$:
  Test whether $P_2$ simulates $\rho_1$. If so continue with the next rule. If not, use Algorithm 12 to directly test whether $\rho_1 \preceq^{C}_{P_1} P_2$. If so continue with the next rule; if not halt and return failure.
(ii) Proceed similarly with each Deny rule of $P_2$.
(iii) If no failure is reported above, return success.

**Theorem 17.** *Algorithm 16 is sound and complete for testing* $P_1 \preceq^{C} P_2$.

*Proof.* This follows from Lemmas 11, 13, and 15. □

Algorithm 16 can produce counterexamples when the containment check fails. The check in Algorithm 12 identifies both a request that violates containment (formed from

the head of the rule causing failure) and a path through the dynamic access model to a set of facts that fail to support the request. Counterexamples are important for creating useful analyses, as experience with model checking has shown.

Ideally, however, we would like to go beyond mere containment. A policy author would benefit from knowing the *semantic difference* between two policies, given as the set of all requests whose decisions changed from one policy to the other. Furthermore, these differences should be first-class objects, amenable to querying and verification just as policies are. The ability to analyze differences matters because authors can often state precise expectations of changes even if they cannot state global system properties, as Fisler *et al*. [14] discuss. This is therefore an important problem for future work.

## 4 Related Work

Using state transition systems to model programs guarded by access control policies goes back to Bell and LaPadula [6] and Harrison *et al*. [18]. More recent works support state transitions over richer models of access control and properties beyond safety [3, 17, 25, 31, 32, 36]. Our model is unique in separating the static policy from its dynamic environment. This enables us to consider analyses such as semantic differencing that can meaningfully be applied to the policy alone. This separation also reflects the growing practice of writing policies in a different (domain-specific) language from applications.

Role-based access control (RBAC) [37] offers one form of support for a dynamic environment. The role abstraction allows users to change roles without having to modify the policy. In that sense it does illustrate the principle of a dynamic environment, but it is simply not rich enough to model the multitude of sources of change.

Bertino *et al*.'s TRBAC model captures time-sensitive, role-based access control policies [7]. TRBAC views time in concrete units such as hours and days and supports rule enabling and disabling based on concrete times (such as "give the night nurse permission to check charts at 5pm"). This concrete-time model explicitly elides other aspects of the dynamic environment, such as the passage of time induced by program events, and is thus unsuitable for reasoning about interactions between programs and policies.

Guelev *et al*. reduce access control policies to state machines over propositions by encoding each first-order relational term as a separate proposition [17]. They provide propositional temporal logic verification, but do not consider policy comparison. Abiteboul *et al*. verify FO-LTL properties against web services modeled as graphs over relational facts [2, 13]. Our work includes a model of the facts over time whereas theirs assumes that the facts are arbitrary (within a given schema). Spielmann's work on verifying e-commerce systems has similar limitations relative to our project [41].

Alloy [22] supports reasoning about relational data. Several researchers, including the authors, have tried building policy analysis tools atop Alloy [14, 21, 39], but these all assume non-dynamic environments. Alloy's support for temporal reasoning is limited to properties of small bounded-length paths. Frias *et al*.'s DynAlloy tool extends Alloy to handle dynamic specifications [15], but retains Alloy's bounded path restrictions.

Datalog is the foundation for many access-control and related frameworks [5, 12, 23, 29, 30, 34]. These works support only non-temporal query evaluation, while we are targeting richer analyses. Our use of uniform containment is inspired by results in the

database literature. Shmueli [40] showed that simple containment of Datalog programs is undecidable while Sagiv [35] showed that uniform containment of programs is decidable, building on ideas of Cosmadakis and Kanellakis [10].

Several researchers have also built access-control reasoning tools atop Prolog [17, 26, 38], but their work does not address policy comparison. Weissman and Halpern model policies using the full power of first-order logic [43]. Their criticisms of Datalog-based models for capturing request denial do not apply to our model with an explicit Deny predicate. Verifying a property against a static policy in their model reduces to checking validity of first-order logic formulas; policy comparison would reduce to computing the set of first-order models that satisfy one formula but not another. We thus believe our model provides a better foundation for building usable verification tools.

Given that our model involves both relational terms and a transition system, our analyses require logics that integrate predicate logic and temporal operators. Hodkinson *et al*. have shown [19, 20] that such logics have very bad decidability properties, even when the first-order components are restricted to decidable fragments. For example, the monadic fragment of first-order linear temporal logic is undecidable, even restricted to the 2-variable case. (The one-variable fragment is decidable.) For branching-time logics, even the one-variable monadic fragment is undecidable. These results suggest that checking validity or satisfiability (conventional theorem-proving tasks) to reason about first-order properties of dynamic policies would face severe difficulties. This paper uses models of policy environments to yield decidable analysis questions.

Backes *et al*. [4] propose refinement relations as a means for determining whether one policy contains another, but their work focuses solely on policies and does not account for the impact of the dynamic environment. Fisler *et al*. [14] have implemented both verification and semantic differencing for role-based policies, but their work handles only weaker (propositional rather than relational) policy models and ignores the impact of the dynamic environment.

## 5 Perspective

This work has demonstrated the importance of analyzing access-control policies in the dynamic context in which they evaluate requests. A great deal of the subtlety in this work arises because policies are are not two-valued (i.e., they may respond with "not-applicable"), but as we explain in the Introduction, this complexity is crucial to enable policies to be modular and to properly separate concerns and spheres of influence. This paper routinely employs results and insights from both the database and computer-aided verification literature, and thus highlight synergies between the two; however, the definitions and lemmas relating policy containment and accessibility under policies and contexts demonstrate the subtle ways in which these results interact within a common model. We believe that the notions of uniform contextual containment and equivalence defined in this paper are fundamental concepts for a theory of policies. The work in this paper can be used to analyze any situation where a program's execution is governed by a logical policy, but we have not explored applications other than access control.

*Acknowledgements:* The authors thank Moshe Vardi for a question that inspired many clarifications in the formalization. This work is partially supported by NSF grants.

## References


1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 61(2):236–269, 2000.
3. T. Ahmed and A. R. Tripathi. Static verification of security requirements in role based CSCW systems. In *Symposium on Access Control Models and Technologies*, pages 196–203, 2003.
4. M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In *Symposium on Applied Computing*, pages 375–382, 2004.
5. M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop*, 2004.
6. D. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The Mitre Corporation, 1976.
7. E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and Systems Security*, 4(3):191–233, 2001.
8. E. Bertino, P. Samarati, and S. Jajodia. Authorizations in relational database management systems. In *ACM Conference on Computer and Communications Security*, pages 130–139, 1993.
9. P. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and Systems Security*, 5(1):1–35, 2002.
10. S. Cosmadakis and P. Kanellakis. Functional and inclusion dependencies: A graph theoretic approach. In P. Kanellakis and F. Preparata, editors, *Advances in Computing Research*, volume 3: Theory of Databases, pages 163–185. JAI Press, 1986.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
12. J. DeTreville. Binder: a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 95–103, 2002.
13. A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *ACM Symposium on Principles of Database Systems*, pages 71–82, 2004.
14. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, May 2005.
15. M. F. Frias, J. P. Galeotti, C. G. L. Pombo, and N. M. Aguirre. DynAlloy: upgrading Alloy with actions. In *International Conference on Software Engineering*, pages 442–451, 2005.
16. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.
17. D. P. Guelev, M. D. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *Information Security Conference*, number 3225 in Lecture Notes in Computer Science. Springer-Verlag, 2004.
18. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, Aug. 1976.
19. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000.
20. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *IEEE Symposium on Logic in Computer Science*, pages 393–402, 2002.
21. G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, University of California, Santa Barbara, 2004.

22. D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2000.
23. T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.
24. N. D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975.
25. M. Koch, L. V. Mancini, and F. Parisi-Presicce. Decidability of safety in graph-based models for access control. In *European Symposium on Research in Computer Security*, pages 299–243, 2002.
26. G. Kolaczek. Specification and verification of constraints in role based access control for enterprise security system. In *International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 190–195, 2003.
27. S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, pages 2–16, January 2003.
28. B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, Jan. 1974.
29. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and Systems Security*, 6(1):128–171, 2003.
30. N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Symposium on the Practical Aspects of Declarative Languages*, pages 58–73, 2003.
31. N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, May 2005.
32. N. Li and M. V. Tripunitara. Security analysis in role-based access control. In *ACM Symposium on Access Control Models and Technologies*, 2004.
33. T. Moses. eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, Feb. 2003.
34. A. Pimlott and O. Kiselyov. Soutei, a logic-based trust-management system. In *Functional and Logic Programming*, pages 130–145, 2006.
35. Y. Sagiv. Optimizing datalog programs. In *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.
36. R. Sandhu. The schematic protection model: its definition and analysis for acyclic attenuating systems. *Journal of the ACM*, 35(2):404–432, 1988.
37. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
38. B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security*, pages 1–12, April 2004.
39. A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Symposium on Access Control Models and Technologies*, pages 13–22, 2002.
40. O. Shmueli. Decidability and expressiveness aspects of logic queries. In *ACM Symposium on Principles of Database Systems*, pages 237–249, 1987.
41. M. Spielmann. Verification of relational transducers for electronic commerce. In *ACM Symposium on Principles of Database Systems*, pages 92–103. ACM Press, 2000.
42. M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Symposium on the Theory of Computing*, pages 137–146. ACM, 1982.
43. V. Weissman and J. Halpern. Using first-order logic to reason about policies. In *IEEE Computer Security Foundations Workshop*, pages 187–201, 2003.
44. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *European Symposium on Programming*, pages 204–222, 2003.