A Logic for State-Modifying Authorization Policies

Moritz Y. Becker[1] and Sebastian Nanz[2]

[1] Microsoft Research, Cambridge, CB3 0FB, UK
moritzb@microsoft.com
[2] Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
nanz@imm.dtu.dk

Abstract. We present a logic for specifying policies where access requests can have effects on the authorization state. The logic is semantically defined by a mapping to Transaction Logic. Using this approach, updates to the state are factored out of the resource guard, thus enhancing maintainability and facilitating more expressive policies that take the history of access requests into account. We also present a sound and complete proof system for reasoning about sequences of access requests. This gives rise to a goal-oriented algorithm for finding minimal sequences that lead to a specified target authorization state.

1 Introduction

Trust management [1] has become a leading approach for controlling the access to security critical services in distributed environments. One of the reasons for this success is the flexibility and expressiveness provided by their associated authorisation query languages for policy specification. As depicted in Figure 1, these languages allow the authorization policy to be factored out of the hard-coded resource guard and written explicitly as a list of declarative rules. When a principal requests access, the resource guard issues an authorization query to the policy evaluator. Access is granted only if the policy evaluator succeeds in proving that the request complies with the local policy and the authorization state. The latter is a database containing relevant environmental facts including knowledge obtained from (submitted or fetched) credentials.

This approach hugely increases the maintainability of complex systems, as modifying the declarative policy rules is much simpler than rewriting and recompiling pieces of procedural code hidden in the resource guard. However, often the resource guard will not only allow or deny access, but also update the authorization state after a successful request. In a role-based policy, for instance, the fact that a user has activated some role is inserted into the authorization state after a successful role activation request. Similarly, the fact may be removed from the state if the role is deactivated. There are many policies that depend on past interactions; relevant events must therefore be stored in the authorization state. Consider for example the following scenario, where a company policy

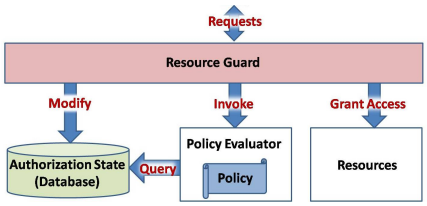

Fig. 1. Model of a policy-based authorization system

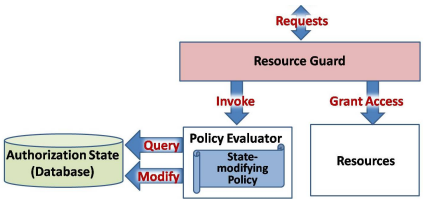

Fig. 2. Factoring out the state manipulations.

specifies that payments are only executed if they are initiated and authorized by two different managers. The resource guard could then contain the following two procedures to implement this policy:

```
procedure initPay(X:Principal, P:Payment) {
  if query(isMgr(X)) and not(query(hasBeenInit(P))) then {
       insertFact(hasBeenInit(P));
       insertFact(hasInitPay(X, P))
     }
}
procedure authPay(X:Principal, P:Payment) {
  if query(isMgr(X)) and query(hasBeenInit(P)) and
     not(query(hasInitPay(X, P))) then
       insertFact(hasBeenAuth(P))
```

If user A attempts to initiate a payment P, the resource guard executes the procedure `initPay(A, P)`, which issues queries to the policy evaluator to check whether A is a manager and that the payment has not already been initiated. If successful, the facts `hasBeenInit(P)` and `hasInitPay(A, P)` are inserted into the authorization state. A request of user B to authorize the same payment P is authorized if B can be proved to be a manager and that the payment has been initiated by someone else.

This example highlights a deficiency in current authorization languages: they cannot express updates of the authorization state, as required for many role-based, separation-of-duties and other history-dependent policies. Instead, updates have to be hard-coded into the resource guard, which leads to maintainability problems. Moreover, as the state changes happen outside the policy and are written in a Turing-complete language, rigorous analysis is difficult.

In this paper, we address this problem by introducing *SMP*, a logic for specifying policies with state-modifying user requests. State changes are thus factored out of the resource guard, as in Figure 2. For example, in our approach the above scenario could be modelled by the following two policy rules:

$$\begin{aligned} \mathsf{initPay}(X,P) \leftarrow\ & \mathsf{isMgr}(X) \wedge \neg\mathsf{hasBeenInit}(P) \otimes \\ & +\mathsf{hasBeenInit}(P) \otimes +\mathsf{hasInitPay}(X,P) \\ \mathsf{authPay}(X,P) \leftarrow\ & \mathsf{isMgr}(X) \wedge \mathsf{hasBeenInit}(P) \wedge \neg\mathsf{hasInitPay}(X,P) \otimes \\ & +\mathsf{hasBeenAuth}(P) \end{aligned}$$

Intuitively, $+P$ specifies an insertion of a fact P into the authorization state, and the connective $\otimes$ expresses a serial execution of state changes.

The complexity of state-modifying policies calls for analysis tools that support policy authors in debugging policies. We present a proof system that describes all possible sequences of access requests which yield a certain outcome. This proof system is proved sound and correct with respect to the logic, and we describe a sound and complete algorithm for finding minimal sequences in the propositional case.

The remainder of this paper is structured as follows. In §2 we introduce *SMP* and motivate its syntax and semantics. In §3 we present a proof system for reasoning about sequences of access requests, and describe the related algorithm. In §4 we present a larger case study for using state-modifying authorization policies. We conclude in §5 with a discussion on related works. A technical report [2] contains full proofs.

2 A Logic for State-Modifying Policies

This section introduces the syntax and semantics of *SMP*. This logic is based on Datalog, but extends it with statements for state modification, henceforth called *effects*, and a simple form of negation. We assume a denumerable set of variables $\mathcal{X}$ and a first-order signature with constants $\mathcal{C}$ and disjoint sets of predicate symbols, *extensional* ($\mathcal{Q}_{ext}$) and *intensional* ($\mathcal{Q}_{int}$) ones, as in standard Datalog. In addition, we have a third set of so-called *command* predicate symbols $\mathcal{Q}_{cmd}$, intended to represent access requests. As usual, (extensional, intensional and command) *atoms* are formed by applying predicate symbols to ordered lists of constants or variables. A *literal* is either an atom P or a negated atom $\neg P$.

The extensional predicates are defined by an *extensional database* (EDB), a set of ground extensional atoms (*facts*). The validity of an extensional literal can thus be checked simply by inspecting the database. In the context of an authorization system, the database contains environmental facts that are relevant for authorization, e.g. $\mathsf{hasInitPay}(\mathtt{Alan}, \mathtt{P123})$ or $\mathsf{isUser}(\mathtt{Alan})$. As we shall see later, facts may be inserted or removed from the database as the result of evaluating an access request. The database thus constitutes the transient state of an authorization system; hence we also call it the *authorization state*.

Definition 2.1. An *authorization state* $\mathbf{B}$ is a finite set of *facts*.

Intensional predicate symbols are defined by *rules*. A rule consists of an intensional atom P_{int} (the *head*) and a conditional *body*, a (possibly empty) conjunction of extensional or intensional literals: $P_{int} \leftarrow L_1 \wedge \ldots \wedge L_m$. For example, the intensional predicate symbol isMgr may be defined by a rule specifying that X is a manager if X is a user and if someone has registered X as a manager:

$$\mathsf{isMgr}(X) \leftarrow \mathsf{isUser}(X) \wedge \mathsf{hasRegisteredAsMgr}(Y, X)$$

Negation is restricted to extensional atoms: $\neg P_{ext}$ holds if P_{ext} is not in the current authorization state. This is the simplest form of negation that is sufficient for our purposes.

Term	$t ::= X \mid a$	where $X \in \mathcal{X}, a \in \mathcal{C}$
Atom	$P_\tau ::= p(t_1, \ldots, t_k)$	where $p \in \mathcal{Q}_\tau,\ \tau \in \{ext, int, cmd\}$
Literal	$L ::= P_{int} \mid P_{ext} \mid \neg P_{ext}$	
Effect	$K ::= +P_{ext} \mid -P_{ext}$	
Rule	$Rl ::= P_{int} \leftarrow L_1 \wedge \ldots \wedge L_m$	where $m \geq 0$
	$\mid P_{cmd} \leftarrow L_1 \wedge \ldots \wedge L_m \otimes K_1 \otimes \ldots \otimes K_n$	where $m, n \geq 0$

Table 1. Syntax of *SMP*

Access requests, or *commands*, are defined by *command rules*, i.e. rules with a command atom as head. In addition to the body, a command rule contains a (possibly empty) sequence of *effects* on the authorization state which are executed if all conditions in the body have been satisfied. An effect is either an insertion ($+P_{ext}$) of a fact P_{ext} into the authorization state or a removal ($-P_{ext}$). Effects are sequentially composed by the operator $\otimes$ from Transaction Logic. A command rule is thus of the form $P_{cmd} \leftarrow L_1 \wedge \ldots \wedge L_m \otimes K_1 \otimes \ldots \otimes K_n$. We sometimes write $\vec{L}$ to abbreviate $L_1 \wedge \ldots \wedge L_m$, and $\tilde{K}$ for $K_1 \otimes \ldots \otimes K_n$. Table 1 shows the complete *SMP* syntax.

Definition 2.2 (Well-formed Policy). A *policy* is a finite set of rules. A rule is *well-formed* if all variables of its effects also occur in the head; furthermore, if effects $+P_1$ and $-P_2$ occur in the same rule, then P_1 and P_2 are non-unifiable. A policy $\mathcal{P}$ is *well-formed* iff all rules in $\mathcal{P}$ are well-formed, and furthermore, whenever two ground instances of rules in $\mathcal{P}$ have the same head, their effects are identical.

The well-formedness conditions ensure that every ground command uniquely determines a sequence of ground effects, and furthermore, that the order of the sequence is irrelevant. As command literals cannot occur inside a body, effects can only occur at the top level: they are effectively decoupled from recursion. Recursive effects would not only be harder to compute and to comprehend, but worse, they would be non-deterministic.

Example 2.3. *We would like to write a policy for an online movie store. Informally, we would like to express that users can buy a movie online, and are then allowed to play it twice. This is expressed by the following transaction policy:*

$$\begin{aligned}
\mathsf{buy}(X, M) &\leftarrow +\mathsf{bought}(X, M) \\
\mathsf{play1}(X, M) &\leftarrow \mathsf{bought}(X, M) \wedge \neg\mathsf{played1}(X, M) \otimes +\mathsf{played1}(X, M) \\
\mathsf{play2}(X, M) &\leftarrow \mathsf{played1}(X, M) \wedge \neg\mathsf{played2}(X, M) \otimes +\mathsf{played2}(X, M)
\end{aligned}$$

The semantics of *SMP* is formalized by modelling it as a fragment of Transaction Logic [3]. Transaction Logic is a general framework that incorporates database updates and transactions into first order logic. A Herbrand-style model theory of Transaction Logic is presented in detail in [3]. Based on this semantics, we define an entailment relation $\tilde{\mathbf{B}} \models_{\mathcal{P}} \phi$ between a sequence of authorization

(pos)	$\mathbf{B} \vDash_{\mathcal{P}} P_{ext}$	iff	$P \in \mathbf{B}$
(neg)	$\mathbf{B} \vDash_{\mathcal{P}} \neg P_{ext}$	iff	$P \notin \mathbf{B}$
(and)	$\tilde{\mathbf{B}} \vDash_{\mathcal{P}} \phi \wedge \psi$	iff	$\tilde{\mathbf{B}} \vDash_{\mathcal{P}} \phi$ and $\tilde{\mathbf{B}} \vDash_{\mathcal{P}} \psi$
(seq)	$\mathbf{B}_1, ..., \mathbf{B}_k \vDash_{\mathcal{P}} \phi \otimes \psi$	iff	$\mathbf{B}_1, ..., \mathbf{B}_i \vDash_{\mathcal{P}} \phi$ and $\mathbf{B}_i, ..., \mathbf{B}_k \vDash_{\mathcal{P}} \psi$ for some $i \in \{1, ..., k\}$
(plus)	$\mathbf{B}_1, \mathbf{B}_2 \vDash_{\mathcal{P}} +P$	iff	$\mathbf{B}_2 = \mathbf{B}_1 \cup \{P\}$
(min)	$\mathbf{B}_1, \mathbf{B}_2 \vDash_{\mathcal{P}} -P$	iff	$\mathbf{B}_2 = \mathbf{B}_1 \setminus \{P\}$
(impl)	$\tilde{\mathbf{B}} \vDash_{\mathcal{P}} Q$	iff	$Q \leftarrow \phi$ is a ground instantiation of a rule in $\mathcal{P}$ and $\tilde{\mathbf{B}} \vDash_{\mathcal{P}} \phi$ where Q is not extensional

Table 2. *SMP* semantics

states $\tilde{\mathbf{B}}$ and a formula ϕ, in the context of a well-formed policy $\mathcal{P}$. This relation is presented in Table 2.

Intuitively, $\mathbf{B}_0, \ldots, \mathbf{B}_n \vDash_{\mathcal{P}} \phi$ means that the goal ϕ can be derived in the context of policy $\mathcal{P}$, starting from an initial authorization state $\mathbf{B}_0$. The evaluation of ϕ leads (via the intermediate states) to a final state $\mathbf{B}_n$. Rule (pos) and (neg) state that extensional literals are checked by inspecting the authorization state. This involves no effects, so the initial and final state are both identical. Rule (and) states that a database sequence $\tilde{\mathbf{B}}$ entails the conjunction of two formulae ϕ and ψ iff each of them is independently entailed by the same $\tilde{\mathbf{B}}$. In contrast, a serial composition $\phi \otimes \psi$ is entailed by $\mathbf{B}_1, ..., \mathbf{B}_k$ iff ϕ can be derived starting from $\mathbf{B}_1$ and ending in some intermediate state $\mathbf{B}_i$, and ψ can be derived starting from $\mathbf{B}_i$ and ending in $\mathbf{B}_k$. Rules (plus) and (min) straightforwardly describe the insertion and deletion of facts. Finally, rule (impl) defines the derivation for non-extensional literals.

3 Reasoning about User Requests

The increased complexity of state-modifying policies calls for proof techniques and tools to support the policy writer in establishing the correctness of a policy. For instance, in Example 2.3 a policy writer might like to determine the answers to the questions "Is there a command sequence that enables a user to play a movie without purchase?" and "Can a movie be played at least twice after purchase?". In this section, we develop a sound and complete proof system which determines the command sequences which yield a certain target authorization state, and also motivates an algorithm for deriving a finite abstraction of all such possible sequences.

3.1 State Constraints

In analyzing state-modifying policies, we are usually not interested in whether a specific authorization state is reachable. Rather, we wish to reason about the reachability of a family of target states that all satisfy some constraint; for example, all states containing some ground instantiation of the atom $\mathsf{played1}(X, M)$

but not the corresponding instantiation of $\mathsf{bought}(X, M)$. This would capture all states in which a movie has been played without purchase.

The following definition allows us to specify classes of authorization states by stating which atoms it must or must not contain.

Definition 3.1 (State Constraints). A *state constraint* D is a set of extensional literals. The notation D^+ (and D^-, respectively) is used to refer to the set of atoms derived from the positive (negative) literals in D. A state $\mathbf{B}$ *satisfies* a ground state constraint D iff $D^+ \subseteq \mathbf{B}$ and $D^- \cap \mathbf{B} = \emptyset$. A state constraint D is *consistent* iff $D^+ \cap D^- = \emptyset$.

For example, the state $\mathbf{B}_0 = \{p(0), q(1), r(2)\}$ satisfies the state constraint $D = \{p(0), q(1), \neg r(1)\}$ because $D^+ = \{p(0), q(1)\} \subseteq \mathbf{B}_0$ and $D^- = \{r(1)\} \cap \mathbf{B}_0 = \emptyset$. In this way D characterizes a family of states, namely the set of states that satisfy D. The state constraint $E = \{p(0), q(1), \neg p(0)\}$ is not consistent because $E^+ \cap E^- = \{p(0)\} \neq \emptyset$.

Example 3.2. *Continuing Example 2.3, a policy writer might be interested in establishing that there is no command sequence which leads to a state satisfying the constraint* $\{\neg\mathsf{bought}(X, M), \mathsf{played1}(X, M)\}$*; we will establish later that there exists no such sequence. On the other hand, we will be able to determine that the command sequence* $\mathsf{buy}(X, M) \otimes \mathsf{play1}(X, M)$ *leads from an arbitrary state to a state satisfying* $\{\mathsf{bought}(X, M), \mathsf{played1}(X, M)\}$.

Composing state constraints allows us to specify their transformation when a command is performed on them.

Definition 3.3 (Composition Operator). The operator $\circ$ defines a notion of composition for state constraints E and D:

$$(E \circ D)^+ = (D^+ \cup E^+) \setminus E^- \qquad (E \circ D)^- = (D^- \cup E^-) \setminus E^+$$

Intuitively, E can be interpreted as a set of effects, where positive literals in E are interpreted as insertions into D, and negative ones as removals. Then $E \circ D$ can be interpreted as the result of applying the effects in E to D. More precisely: it is the strongest constraint that is satisfied by states obtained by applying E to states satisfying D. For example, let $D = \{p(0), q(1), \neg r(1)\}$ as before and $E = \{\neg p(0), r(1), s(2)\}$ then $E \circ D = \{q(1), r(1), s(2), \neg p(0)\}$. This will enable us later to formulate the strongest postcondition when applying some command Q with effect E to a state constraint D.

Recall that the well-formedness conditions of policies ensure that every ground command atom is uniquely associated with a sequence of its effects. We can now define a mapping *eff* that maps a ground command atom to a state constraint that is equivalent to its effects:

Definition 3.4 (Specification of *eff*). Let Q be a ground command atom, and let $\tilde{K}$ be the (possibly empty) sequence of ground effects uniquely determined by it (in the context of a well-formed policy). Then $\mathit{eff}(Q)$ is defined as the state constraint

$$\mathit{eff}(Q) = \{P \,:\, +P \in \tilde{K}\} \cup \{\neg P \,:\, -P \in \tilde{K}\}$$

3.2 Preconditions and Effects

We wish to relate command sequences of the form $\tilde{S} = Q_1 \otimes \ldots \otimes Q_k$ to state constraints D that represent initial authorization states from which $\tilde{S}$ can successfully execute, and to state constraints D' that represent authorization states after executing the sequence. In other words, for all authorization states $\mathbf{B}$ and $\mathbf{B}'$ such that $\mathbf{B}$ satisfies D and $\mathbf{B}'$ satisfies D', $\mathbf{B}, \ldots, \mathbf{B}' \vDash_{\mathcal{P}} \tilde{S}$ should hold. We start by considering the case $k = 1$, where the command sequence consists of just one single command. The proof system developed in §3.3 then deals with command sequences of arbitrary lengths and captures exactly the above relationship between D, $\tilde{S}$ and D'.

In order to decide from which initial states a command successfully executes, we need to compute its *preconditions* as a state constraint. To determine the target states, we need to find its *effects*. For example, suppose a command predicate q is defined by a single policy rule (where p, r and s are all extensional):

$$q(X, Y) \leftarrow \neg p(X) \wedge r(Z) \otimes +p(X) \otimes -s(Y),$$

Then in order for the command $q(0, 1)$ to execute, the starting authorization state must satisfy the state constraint $\{\neg p(0), r(Z)\}$ containing the preconditions of the command. Likewise, due to the effects of the command, the end state contains $p(0)$ but cannot contain $s(1)$. Moreover, the atom $r(Z)$ is left untouched by the command, hence the end state satisfies $\{r(Z), p(0), \neg s(1)\}$.

The effects are easily obtained using the function *eff* defined in Definition 3.4 that reinterprets the sequence of effects as a state constraint (insertions $+P$ corresponds to positive literals P, and removals $-P$ to negative literals $\neg P$).

In the following, we define a function *pre* which yields the information about the (extensional) preconditions of a ground atom P. In the case where P is entirely defined by rules whose body literals are all extensional, it is easy to determine the preconditions. For instance, in the rule for $\mathsf{play1}(X, M)$ of Example 2.3 both bought and $\mathsf{played1}$ are extensional predicate symbols. Furthermore, $\mathsf{play1}$ is defined by only one single rule. Therefore the preconditions for executing $\mathsf{play1}(X, M)$ are given by a set containing only one state constraint:

$$pre(\mathsf{play1}(X, M)) = \{\{\mathsf{bought}(X, M), \neg\mathsf{played1}(X, M)\}\}$$

In general however, command predicates may be defined by multiple rules, and the body literals of those rules may be intensional, as in the following example.

Example 3.5. *We modify Example 2.3 by removing the* $\mathsf{buy}(X, M)$ *rule, and adding the following two rules (where* bought *is now an intensional predicate):*

$$\mathsf{bought}(X, M) \leftarrow \mathsf{bank}(Y), \mathsf{cardPayment}(X, Y, M)$$
$$\mathsf{bought}(X, M) \leftarrow \mathsf{freeTrial}(X)$$

These rules express that customer X has bought a movie M if he either paid for it using his credit card issued by a bank Y, or has signed up for a free trial offer

of the movie store. In this case, $pre(\mathsf{play1}(X, M))$ *is a set containing the two preconditions under which the command* play1 *can execute:*

$$F_1 = \{\mathsf{bank}(Y), \mathsf{cardPayment}(X, Y, M), \neg\mathsf{played1}(X, M)\}$$
$$F_2 = \{\mathsf{freeTrial}(X), \neg\mathsf{played1}(X, M)\}$$

To specify *pre* formally, note that the bodies of the rules defining P determine its preconditions; in particular, the effects have no influence. We therefore introduce the following operation on policies that erases the effects:

Definition 3.6. Let $\mathcal{P}$ be a well-formed policy, and

$$Rl \equiv P_{cmd} \leftarrow L_1 \wedge \ldots \wedge L_m \otimes E_1 \otimes \ldots \otimes E_n$$

be one of its rules. Then $Rl^\star$ denotes the pure Datalog rule $P_{cmd} \leftarrow L_1 \wedge \ldots \wedge L_m$, where all effects have been removed. We write $\mathcal{P}^\star$ to denote the policy obtained from $\mathcal{P}$ by applying the $\star$-operation to all its rules.

We are now ready to specify *pre*:

Definition 3.7 (Specification of *pre*)**.** The function *pre* is a mapping between ground atoms and sets of state constraints. It is defined by the following axiom which holds for any state $\mathbf{B}$ and ground atom P:

$$\exists\, F, \theta \;:\; F \in pre(P) \wedge \mathbf{B} \text{ satisfies } F\theta \quad \text{iff} \quad \mathbf{B} \vDash_{\mathcal{P}^\star} P$$

Read in the "only if"-direction, the axiom states that every authorization state satisfying a ground instantiation of a state constraint in $pre(P)$ indeed suffices as a precondition of P. In the "if"-direction, the axiom states that every extensional ground precondition is subsumed by some state constraint in $pre(P)$.

Note that literals in preconditions can also contain free variables, such as Y in the first rule of Example 3.5. These variables are instantiated by some substitution θ.

The function *pre* can be computed using *abduction* [4], a dual to *deduction*, where explanatory facts, e.g. F_1 in the example, are inferred from a desired result, e.g. $\mathsf{bought}(X, M)$, using some general (effect-free) logic program, $\mathcal{P}^\star$. Abduction algorithms for logic programs are documented in [5].

3.3 Proof System

Based on the notion of state constraints and the functions *pre* and *eff* defined above, we now present a Hoare-style proof system in which we can reason about pre- and postconditions of general command sequences. The proof system allows the inference of triples of the form $\{D_1\}\,\tilde{S}\,\{D_2\}$. Intuitively, this holds if $\tilde{S}$ can be executed from any authorization state satisfying D_1, and terminates in a state satisfying D_2. The empty command ε trivially transforms any ground and consistent state constraint G into itself, according to rule (eps). The rule (cmd) relates commands to their preconditions and postconditions based on

$$\text{(eps)}\frac{G \text{ is ground and consistent}}{\{G\}\,\varepsilon\,\{G\}} \qquad \text{(seq)}\frac{\{G_1\}\,\tilde{S}\,\{H\} \qquad \{H\}\,P\,\{G_2\}}{\{G_1\}\,\tilde{S} \otimes P\,\{G_2\}}$$

$$\text{(cmd)}\frac{\begin{array}{c} G_1 \supseteq F\theta \qquad F \in \mathit{pre}(P) \qquad \mathit{eff}(P) \circ G_1 \supseteq G_2 \\ G_1 \text{ is ground and consistent} \qquad P \text{ is ground} \end{array}}{\{G_1\}\,P\,\{G_2\}}$$

Table 3. Proof system

pre and *eff*, but also incorporates precondition strengthening and postcondition weakening: a command P transforms G_1 into G_2 if the following holds: there exists a substitution θ such that the free variables in a precondition $F \in \mathit{pre}(P)$ can be resolved with $G_1 \supseteq F\theta$, i.e. $F\theta$ is a weakest precondition for the execution of P and G_1 is strong enough to satisfy this preconditon; furthermore, G_2 has to be contained in $\mathit{eff}(P) \circ G_1$, i.e. G_2 is weaker than the *strongest postcondition* obtained by computing $\mathit{eff}(P) \circ G_1$. Finally, rule (seq) states that a sequence $\tilde{S}$ and a command P can be executed sequentially, if the postcondition of $\tilde{S}$ is strong enough to serve as a precondition for P.

The correctness of the proof system is ensured by the following soundness and completeness results, which relate it back to *SMP* semantics.

Theorem 3.8 (Soundness). *For all state constraints G, G', command sequences $\tilde{S} \neq \varepsilon$ and authorization states $\mathbf{B}_1$ the following holds. If $\{G\}\,\tilde{S}\,\{G'\}$ and $\mathbf{B}_1$ satisfies G, then there exists a sequence of authorization states $\mathbf{B}_1, ..., \mathbf{B}_n$ (for some $n \geq 1$) such that $\mathbf{B}_1, \ldots, \mathbf{B}_n \vDash_{\mathcal{P}} \tilde{S}$ and $\mathbf{B}_n$ satisfies G'.*

Theorem 3.9 (Completeness). *For all sequences of authorization states $\mathbf{B}_1$, ..., $\mathbf{B}_n$ (where $n \geq 1$), command sequences $\tilde{S}$ and state constraints G' the following holds. If $\mathbf{B}_1, ..., \mathbf{B}_n \vDash_{\mathcal{P}} \tilde{S}$ and $\mathbf{B}_n$ satisfies G', then there exists G such that $\{G\}\,\tilde{S}\,\{G'\}$ and $\mathbf{B}_1$ satisfies G.*

3.4 Tabling Algorithm

The proof system gives rise to an algorithm for computing an abstraction of the set of all sequences which, given an authorization state $\mathbf{B}_0$ to start with, lead to states that are guaranteed to satisfy a target state constraint D. The only inputs to the algorithm are $\mathbf{B}_0$, D, and a well-formed program $\mathcal{P}$. We present this algorithm for the propositional case, i.e. the policy and the state constraints are variable-free. A prototype implementation of the algorithm has been used to confirm the results from the examples presented in this section.

The algorithm uses a goal-oriented search that attempts to construct sequences backwards. To ensure completeness and termination, and to prune the search trees, intermediate results are cached in a table and are reused. The technique of tabling, or memoing [6,7,8,9], has also been considered for policy evaluation [10,11]. Our algorithm works on *nodes* that represent both answers and unsolved goals which arise during the computation.

$$
\begin{array}{lll}
\text{(root)} & (\{\langle D_2\rangle\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow_{\mathcal{P}} (\mathcal{N} \cup \mathcal{N}_1 \cup \mathcal{N}_2, \mathit{Ans}, \mathit{Wait}) & \\
& \text{if} \quad \mathcal{N}_1 = \{\mathsf{goal}\langle D_1; Q; D_2\rangle & : Q \text{ is some command in } \mathcal{P}, \text{ and } E \stackrel{\text{def}}{=} \mathit{eff}(Q) \\
& & \quad\; E \cap D_2 \neq \emptyset \\
& & \quad\; F \in \mathit{pre}(Q) \\
& & \quad\; D_1 \stackrel{\text{def}}{=} (D_2 \setminus E) \cup F \text{ is consistent} \\
& & \quad\; {D_2}^+ \cap E^- = \emptyset \text{ and } {D_2}^- \cap E^+ = \emptyset\} \\
& \quad\; \mathcal{N}_2 = \{\mathsf{ans}\langle D_2; \varepsilon; D_2\rangle & : \mathbf{B}_0 \text{ satisfies } D_2\}
\end{array}
$$

$$
\begin{array}{lll}
\text{(ans)} & (\{\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow_{\mathcal{P}} (\mathcal{N} \cup \mathcal{N}', \mathit{Ans}', \mathit{Wait}) & \\
& \text{if} \quad \nexists\, m \in \mathit{Ans}(D_2) \;:\; \mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle \preceq m & \\
& \quad\; \mathcal{N}' = \bigcup_{n' \in \mathit{Wait}(D_2)} \mathit{merge}(\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle, n') & \\
& \quad\; \mathit{Ans}' = \mathit{Ans}[D_2 \mapsto \mathit{Ans}(D_2) \cup \{\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle\}] &
\end{array}
$$

$$
\begin{array}{lll}
\text{(goal}_1\text{)} & (\{\mathsf{goal}\langle D_1; Q; D_2\rangle\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow_{\mathcal{P}} (\mathcal{N} \cup \mathcal{N}', \mathit{Ans}, \mathit{Wait}') & \\
& \text{if} \quad D_1 \in \mathit{dom}(\mathit{Ans}) & \\
& \quad\; \mathcal{N}' = \bigcup_{n' \in \mathit{Ans}(D_1)} \mathit{merge}(n', \mathsf{goal}\langle D_1; Q; D_2\rangle) & \\
& \quad\; \mathit{Wait}' = \mathit{Wait}[D_1 \mapsto \mathit{Wait}(D_1) \cup \{\mathsf{goal}\langle D_1; Q; D_2\rangle\}] &
\end{array}
$$

$$
\begin{array}{lll}
\text{(goal}_2\text{)} & (\{\mathsf{goal}\langle D_1; Q; D_2\rangle\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow_{\mathcal{P}} (\mathcal{N} \cup \{\langle D_1\rangle\}, \mathit{Ans}', \mathit{Wait}') & \\
& \text{if} \quad D_1 \notin \mathit{dom}(\mathit{Ans}) & \\
& \quad\; \mathit{Ans}' = \mathit{Ans}[D_1 \mapsto \emptyset] & \\
& \quad\; \mathit{Wait}' = \mathit{Wait}[D_1 \mapsto \{\mathsf{goal}\langle D_1; Q; D_2\rangle\}] &
\end{array}
$$

Table 4. Tabling algorithm

Definition 3.10 (Nodes). A *root node* is of the form $\langle D\rangle$ where D is a state constraint. An *answer node* is denoted $\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle$, where $\tilde{T}$ is a sequence and D_1 and D_2 are state constraints, called the *pre-* and *postcondition* of $\tilde{T}$, respectively. The same terms apply to a *goal node* $\mathsf{goal}\langle D_1; \tilde{T}; D_2\rangle$.

Intuitively, if $\langle D_2\rangle$ occurs in the node set the algorithm is working on, we are looking for command sequences which have D_2 as final state. Tables *Ans* and *Wait* are defined to map state constraints into sets of answer and goal nodes, respectively. If $\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle \in \mathit{Ans}(D_2)$, then $\tilde{T}$ is an answer to D_2, i.e. $\mathbf{B}_0$ satisfies D_1, and $\{D_1\}\,\tilde{T}\,\{D_2\}$. If $\mathsf{goal}\langle D_1; \tilde{T}; D_2\rangle \in \mathit{Wait}(D_1)$, then $\tilde{T}$ is not yet an answer to D_2, but it is waiting for new answers to D_1 which can be resolved with the goal to give a new answer to D_2.

We present the algorithm as a state transition system in Table 4. A state is given by the current node set, the answer table, and the waiting table together.

Definition 3.11 (State). A *state* is a triple $(\mathcal{N}, \mathit{Ans}, \mathit{Wait})$ where $\mathcal{N}$ is a set of nodes, *Ans* is an answer table, and *Wait* is a wait table. For every consistent state constraint D, a state $(\{D\}, [D \mapsto \emptyset], [D \mapsto \emptyset])$ is an *initial state*. A state $\mathcal{S}$ is a *final state* iff there is no state $\mathcal{S}'$ such that $\mathcal{S} \rightarrow_{\mathcal{P}} \mathcal{S}'$.

When a new root $\langle D_2\rangle$ is spawned, it is processed by transition (root). If $\mathbf{B}_0$ satisfies D_2, then a single answer node $\mathsf{ans}\langle D_2; \varepsilon; D_2\rangle$ is produced (set $\mathcal{N}_2$), corre-

sponding to rule (eps) of the proof system. Set $\mathcal{N}_1$ contains goal nodes, which are produced according to the following scheme which corresponds to rule (cmd): for all commands Q, it is tested whether they can contribute at least one effect which occurs in D_2, by checking $E \cap D_2 \neq \emptyset$. Then for all preconditions $F \in \mathit{pre}(Q)$, a constraint D_1 is computed, which is obtained by removing all effects of Q from D_2 and adding the precondition F, thus reasoning *backwards* to a state, starting from which Q can execute and yield D_2. The constraint D_1 has to be consistent, and so must be the union $D_2 \cup E$, which is ensured by two further side conditions.

Before explaining the remaining rules, we need two further definitions. The first one deals with deriving new answers from given answers and corresponding goals.

Definition 3.12 (Merge). An answer node $n_1 \equiv \mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle$ and a goal node $n_2 \equiv \mathsf{goal}\langle E_1; Q; E_2\rangle$ can be *merged* iff $D_2 = E_1$ and D_1, D_2, E_1, E_2 are all consistent. The resulting node is $n \equiv \mathsf{ans}\langle D_1; \tilde{T} \otimes Q; E_2\rangle$, and we write $n = \mathit{merge}(n_1, n_2)$.

In general, given an initial state $\mathbf{B}_0$ and a target state constraint D, there are infinitely many correct command sequences that lead from $\mathbf{B}_0$ to D. However, most of these answers are redundant in the sense that there is a shorter one consisting of the same atoms, or there is one with the same length but involves a smaller number of different commands. It turns out that, even if the complete set of answers is infinite, the set of "minimal" answers is finite. The presented algorithm always terminates and is complete with respect to this finite set of answers. All other answers are then *subsumed* by one of the computed answers.

The following definition defines the subsumption relation between answers. In this definition, let *atoms* be a function such that $\mathit{atoms}(Q_1 \otimes \cdots \otimes Q_k) = \{Q_1, ..., Q_k\}$ and $\mathit{atoms}(\varepsilon) = \emptyset$.

Definition 3.13 (Subsumption for Answers). Let $n \equiv \mathsf{ans}\langle D_1; \tilde{S}; D_2\rangle$ and $m \equiv \mathsf{ans}\langle E_1; \tilde{T}; E_2\rangle$. Then, n is said to be *subsumed by* m, written $n \preceq m$, iff

$$D_1 \supseteq E_1 \ \wedge \ D_2 = E_2 \ \wedge \ |\tilde{S}| \geq |\tilde{T}| \ \wedge \ \mathit{atoms}(\tilde{S}) \supseteq \mathit{atoms}(\tilde{T})$$

The idea is that node n is subsumed by m if m is a "better" answer: that is, n's precondition is at least as strong as m's, its postcondition is equal to m's, and its command sequence is at least as long as m's and involves the same (or more) atoms.

Answer nodes are processed by rule (ans). The rule executes if the answer node in question is not subsumed by another one already in $\mathit{Ans}(D_2)$. Then the node is added to the answer table, and it is merged with all waiting goals in $\mathit{Wait}(D_2)$. Goal nodes are processed by either (goal_1) or (goal_2), according to whether or not their precondition D_1 has been spawned before. If not, (goal_2) ensures that $\langle D_1\rangle$ is spawned and that the answer and waiting tables are properly initialized for this new goal. Otherwise, (goal_1) prescribes that the goal node is merged with all answer nodes that might be already available for D_1, and that

the waiting table is updated. The companion technical report [2] has a worked example demonstrating the functionality of the algorithm.

The correctness of the algorithm is stated in the following soundness and completeness theorems:

Theorem 3.14 (Soundness). *If* $(\mathcal{N}, Ans, Wait)$ *is reachable from some initial state, then the following holds for all* $D_2 \in dom(Ans)$*: if* $\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle \in Ans(D_2)$ *and* D_1 *is consistent, then* $\{D_1\}\,\tilde{T}\,\{D_2\}$ *and* $\mathbf{B}_0$ *satisfies* D_1*.*

Intuitively, the soundness theorem states that every answer produced by the algorithm is indeed a correct answer with respect to the proof system. By soundness of the proof system, the algorithm is then also sound with respect to the *SMP* semantics.

Theorem 3.15 (Completeness). *If* $(\mathcal{N}, Ans, Wait)$ *is a final state reachable from some initial state, the following implication holds for all* $D_2 \in dom(Ans)$*: if* $\{G_1\}\,\tilde{S}\,\{G_2\}$*,* $G_2 \supseteq D_2$*, and* $\mathbf{B}_0$ *satisfies* G_1 *then there exists* $\mathsf{ans}\langle D_1; \tilde{T}; D_2\rangle \in Ans(D_2)$ *such that* $\mathbf{B}_0$ *satisfies* D_1*,* $|\tilde{S}| \geq |\tilde{T}|$*, and* $atoms(\tilde{S}) \supseteq atoms(\tilde{T})$*.*

The completeness theorem states that if a sequence is a correct answer with respect to the proof system, then the algorithm produces an answer that is at least as good, in the sense that it may be shorter or involve a smaller number of different commands.

Finally, the following theorem proves that the algorithm always terminates.

Theorem 3.16 (Termination). *All transition paths starting from an initial state are of finite length.*

4 Case Study

This section presents a larger, non-trivial example policy from the area of electronic health care, based on [12]. The policy defines roles such as patient, clinician, or administrator. Users may be members of several roles, and may activate such roles within a session, in order to utilize the privileges associated with the roles. In the following example, a user may activate the patient role with the command activate. The command succeeds if the user is a member of that role (expressed by the predicate member), and as a result, a corresponding hasActivated fact is inserted into the authorization state. Patients can deactivate their own role if this fact is in the state. The deactivation entails the removal of the fact from the state.

$$\mathsf{activate}(X, \texttt{Patient}) \leftarrow \mathsf{member}(X, \texttt{Patient}) \otimes +\mathsf{hasActivated}(X, \texttt{Patient})$$
$$\mathsf{deactivate}(X, \texttt{Patient}) \leftarrow \mathsf{hasActivated}(X, \texttt{Patient}) \otimes -\mathsf{hasActivated}(X, \texttt{Patient})$$

For some groups of roles, the policy specifies a separation-of-duties constraint: users may be active in at most one of the roles at the same time. In the rule below, the clinician role may only be activated if the user is not already active in the administrator role. We also have a symmetric rule for administrators where

"`Clinician`" and "`Admin`" are permuted. The deactivation rules for clinicians and administrators are similar to the one for patients above.

$$\mathsf{activate}(X, \texttt{Clinician}) \leftarrow \mathsf{member}(X, \texttt{Clinician}) \wedge \neg\mathsf{hasActivated}(X, \texttt{Admin}) \otimes \\ +\mathsf{hasActivated}(X, \texttt{Clinician})$$

In the examples above, role membership is a prerequisite for role activation. The following two clauses let administrators update the role membership assignment, using the commands register and unregister.

$$\begin{aligned} \mathsf{register}(X, U, R) &\leftarrow \mathsf{hasActivated}(X, \texttt{Admin}) \otimes +\mathsf{member}(U, R) \\ \mathsf{unregister}(X, U, R) &\leftarrow \mathsf{hasActivated}(X, \texttt{Admin}) \wedge \mathsf{registered}(U, R) \otimes \\ &\quad -\mathsf{member}(U, R) \otimes -\mathsf{hasActivated}(U, R) \end{aligned}$$

Permission assignments typically do not manipulate the authorization state, but often specify conditions that depend on the state. The following clause is an example of a deny-override policy: it allows a clinician X to read data from a patient P's health record if X has a so-called legitimate relationship with the patient, and if the patient has not explicitly concealed the record from X.

$$\mathsf{permitted}(X, \texttt{Read}, P) \leftarrow \mathsf{hasActivated}(X, \texttt{Clinician}) \wedge \mathsf{legitRelationship}(X, P) \wedge \\ \neg\mathsf{denied}(P, X)$$

The command readEHR is executed when a user X attempts to read a patient P's electronic health record (EHR). The read access is stored in the authorization state for auditing purposes.

$$\mathsf{readEHR}(X, P) \leftarrow \mathsf{permitted}(X, \texttt{Read}, P) \otimes +\mathsf{hasReadEHR}(X, P)$$

Patients can conceal data from a clinician using the command denyAccess, and remove the concealment with removeDenyAccess. The two corresponding clauses insert (or remove) a denied fact from the authorization state.

$$\begin{aligned} \mathsf{denyAccess}(P, X) &\leftarrow \mathsf{hasActivated}(P, \texttt{Patient}) \otimes +\mathsf{denied}(P, X) \\ \mathsf{removeDenyAccess}(P, X) &\leftarrow \mathsf{hasActivated}(P, \texttt{Patient}) \wedge \mathsf{denied}(P, X) \otimes \\ &\quad -\mathsf{denied}(P, X) \end{aligned}$$

Access to patient data is conditioned on a *legitimate relationship* between the requester and the patient. The following clause specifies that a legitimate relationship exists between a clinician X and a patient P if the patient has explicitly consented to treatment:

$$\mathsf{legitRelationship}(X, P) \leftarrow \mathsf{hasConsented}(P, X, \texttt{Treatment})$$

The following clauses manage the updates for hasConsented facts. Consent is here modelled as a two-step process: a clinician can request consent to treatment, and only then can the patient give consent.

$$\begin{aligned} \mathsf{giveConsent}(P, X, \texttt{Treatment}) &\leftarrow \mathsf{hasActivated}(P, \texttt{Patient}) \wedge \\ &\quad \mathsf{hasRequestedConsent}(X, P) \otimes \\ &\quad +\mathsf{hasConsented}(P, X, \texttt{Treatment}) \\ \mathsf{requestConsent}(X, P, \texttt{Treatment}) &\leftarrow \mathsf{hasActivated}(X, \texttt{Clinician}) \otimes \\ &\quad +\mathsf{hasRequestedConsent}(X, P, \texttt{Treatment}) \end{aligned}$$

Finally, patients can withdraw their consent to treatment. Similarly, clinicians can cancel the treatment. The command cancelTreatment removes both the consent request and the (possibly non-existing) patient consent fact from the authorization state.

$$\begin{aligned}\mathsf{withdrawConsent}(P, X, \mathtt{Treatment}) \leftarrow\ & \mathsf{hasActivated}(P, \mathtt{Patient}) \wedge \\ & \mathsf{hasConsented}(P, X, \mathtt{Treatment}) \otimes \\ & -\mathsf{hasConsented}(P, X, \mathtt{Treatment})\end{aligned}$$

$$\begin{aligned}\mathsf{cancelTreatment}(X, P) \leftarrow\ & \mathsf{hasActivated}(X, \mathtt{Clinician}) \otimes \\ & -\mathsf{hasRequestedConsent}(X, P, \mathtt{Treatment}) \otimes \\ & -\mathsf{hasConsented}(P, X, \mathtt{Treatment})\end{aligned}$$

Suppose we start in an authorization state in which we know that Alice (A) is a member of the administrator role and has not activated the clinician role. Furthermore, Bob (B) has not explicitly concealed his data from Alice. We can then infer a command sequence that terminates in a state where Alice has read Bob's EHR:

$$\begin{aligned}&\{\mathsf{member}(\mathtt{A}, \mathtt{Admin}), \neg\mathsf{hasActivated}(\mathtt{A}, \mathtt{Clinician}),\ \neg\mathsf{denied}(\mathtt{B}, \mathtt{A})\} \\ &\quad \mathsf{activate}(\mathtt{A}, \mathtt{Admin}) \otimes \mathsf{register}(\mathtt{A}, \mathtt{A}, \mathtt{Clinician}) \otimes \mathsf{register}(\mathtt{A}, \mathtt{B}, \mathtt{Patient}) \otimes \\ &\quad \mathsf{activate}(\mathtt{B}, \mathtt{Patient}) \otimes \mathsf{deactivate}(\mathtt{A}, \mathtt{Admin}) \otimes \mathsf{activate}(\mathtt{A}, \mathtt{Clinician}) \otimes \\ &\quad \mathsf{requestConsent}(\mathtt{A}, \mathtt{B}, \mathtt{Treatment}) \otimes \mathsf{giveConsent}(\mathtt{B}, \mathtt{A}, \mathtt{Treatment}) \otimes \mathsf{readEHR}(\mathtt{A}, \mathtt{B}) \\ &\{\mathsf{hasReadEHR}(\mathtt{A}, \mathtt{B})\}\end{aligned}$$

5 Discussion

Related work Cassandra [10] is an authorization language that defines the actions of activating a role and deactivating a role, along with a transition system that updates the authorization state by inserting and removing corresponding "hasActivated" facts. Users can thus write state-dependent and implicitly state-manipulating policies, but this rather ad-hoc approach is inflexible and not very user-friendly. In a similar spirit, dynFAF [13] keeps track of the history of user requests by dynamically adding facts (with a time-stamp parameter) to the logic program. In dynFAF, facts are never removed; instead, permissions are signed, and permission revocation is modelled by adding a fact with a negative permission. In [14], a sub-language of Timed Default Concurrent Constraint Programming is used to specify dynamic policies. Their language, being almost a full-fledged procedural programming language, can express state changes triggered by both user requests as well as environmental changes. This high expressiveness comes at a price: policies are generally harder to analyze, and evaluation may not terminate.

Some languages such as Ponder [15] or XACML [16] support *obligation policies*. An obligation is a task to be executed after evaluating and enforcing an access request. Obligations are typically used for post-processing jobs such as auditing or for sending out notifications, but in principle an obligation could also be a call to an external function that updates the state. While this approach

would move the effects from the hard-coded resource guard into the policy, it does not provide a precise semantics for the state changes.

Some work has been done on analyzing security properties in dynamic role-based systems, in the context of the role-based authorization language RT [17,18] and Administrative RBAC (ARBAC) [19], where members of administrative roles can modify the role membership and privilege assignments [20]. Security properties in the context of SPKI/SDSI certificates are analyzed in [21] by model checking pushdown automata. [22] presents a Datalog-based logical framework for representing and reasoning about access control policies. Neither of the two papers deals with policies updating the authorization state.

Changes to the policy itself can obviously affect the set of actions that are permitted or denied. Margrave [23] is a tool that can compute the consequences of changes to an XACML policy. Pucella and Weissman [24] consider systems in which policies (not just the facts) can change between state transitions. They introduce a modal logic that can capture the dynamic nature of such systems and prove its decidability in the propositional case. In [25], policies written in Datalog can refer to facts in the authorization state, as in our model. Events (such as access requests) can change the authorization state, and the changes are specified as a state machine whose transition labels are guarded by the policy. Security properties can then be analyzed by model checking formulas in first-order temporal logic.

Conclusion In this paper, we have introduced *SMP*, a logic that not only expresses authorization conditions but can also specify effects of access requests on the authorization state. The effects are specified explicitly in the language (as opposed to e.g. a state machine). The logic can be seen as a mild non-monotonic extension of Datalog and has a formal semantics based on Transaction Logic. Existing authorization languages, especially Datalog-based ones, can thus be easily extended to support effects. Examples of *SMP*'s applicability have been shown in a case study on a policy for electronic health records. We have also presented an inference system for reasoning about sequences of user actions, and a sound and complete goal-oriented algorithm for computing minimal sequences (or proving their non-existence) in the propositional case.

References

1. Blaze, M., Feigenbaum, J., Keromytis, A.D.: The role of trust management in distributed systems security. In: Secure Internet Programming. (1999) 185–210
2. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. Technical Report MSR-TR-2007-32, Microsoft Research (2007)
3. Bonner, A.J., Kifer, M.: An overview of transaction logic. Theoretical Computer Science **133**(2) (1994) 205–265
4. Shanahan, M.: Prediction is deduction but explanation is abduction. In: International Joint Conference on Artificial Intelligence. (1989) 1055–1060
5. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. Journal of Logic and Computation **2**(6) (1992) 719–770

6. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Third international conference on logic programming. (1986) 84–98
7. Dietrich, S.W.: Extension tables: Memo relations in logic programming. In: Symposium on Logic Programming. (1987) 264–272
8. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM **43**(1) (1996) 20–74
9. Toman, D.: Memoing evaluation for constraint extensions of Datalog. Constraints **2**(3/4) (1997) 337–359
10. Becker, M.Y., Sewell, P.: Cassandra: Flexible trust management, applied to electronic health records. In: 17th IEEE Computer Security Foundations Workshop (CSFW). (2004) 139–154
11. Becker, M.Y., Fournet, C., Gordon, A.D.: Design and semantics of a decentralized authorization language. In: Computer Security Foundations Symposium. (2007)
12. Becker, M.Y.: Information governance in NHS's NPfIT: A case for policy specification. International Journal of Medical Informatics **76**(5-6) (2007)
13. Chen, S., Wijesekera, D., Jajodia, S.: Incorporating dynamic constraints in the flexible authorization framework. In: 9th European Symposium on Research Computer Security (ESORICS). (2004) 1–16
14. Jagadeesan, R., Marrero, W., Pitcher, C., Saraswat, V.: Timed constraint programming: a declarative approach to usage control. In: International Conference on Principles and Practice of Declarative Programming. (2005) 164–175
15. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In: International Workshop on Policies for Distributed Systems and Networks. (2001) 18–38
16. OASIS: eXtensible Access Control Markup Language (XACML) Version 2.0 core specification. (2005)
17. Li, N., Tripunitara, M.V.: Security analysis in role-based access control. In: Symposium on Access Control Models and Technologies. (2004) 126–135
18. Li, N., Mitchell, J.C., Winsborough, W.H.: Beyond proof-of-compliance: security analysis in trust management. Journal of the ACM **52**(3) (2005) 474–514
19. Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.R.: Policy analysis for administrative role based access control. In: Workshop on Computer Security Foundations. (2006) 124–138
20. Sandhu, R., Bhamidipati, V., Coyne, E., Canta, S., Youman, C.: The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In: 2nd ACM Workshop on Role-Based Access Control. (1997) 41–54
21. Jha, S., Reps, T.: Analysis of SPKI/SDSI certificates using model checking. In: Computer Security Foundations Workshop. (2002)
22. Bertino, E., Catania, B., Ferrari, E., Perlasca, P.: A logical framework for reasoning about access control models. ACM Trans. Inf. Syst. Secur. **6**(1) (2003) 71–127
23. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: 27th International Conference on Software engineering. (2005) 196–205
24. Pucella, R., Weissman, V.: Reasoning about dynamic policies. In: Foundations of Software Science and Computation Structures. (2004) 453–467
25. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: Automated Reasoning, Third International Joint Conference, IJCAR 2006. (2006) 632–646