A Logic-based Framework for Attribute based Access Control*

Lingyu Wang, Duminda Wijesekera† and Sushil Jajodia
Center for Secure Information Systems,
George Mason University, Fairfax VA 22030.
e-mail: {lwang3|dwijesek|jajodia}@gmu.edu

ABSTRACT

Attribute based access control (ABAC) grants accesses to services based on the attributes possessed by the requester. Thus, ABAC differs from the traditional discretionary access control model by replacing the *subject* by a set of attributes and the *object* by a set of services in the access control matrix. The former is appropriate in an identityless system like the Internet where subjects are identified by their characteristics, such as those substantiated by certificates. These can be modeled as attribute sets. The latter is appropriate because most Internet users are not privy to method names residing on remote servers. These can be modeled as sets of service options. We present a framework that models this aspect of access control using logic programming with set constraints of a computable set theory [DPPR00]. Our framework specifies policies as stratified constraint flounder-free logic programs that admit primitive recursion. The design of the policy specification framework ensures that they are consistent and complete. Our ABAC policies can be transformed to ensure faster runtimes.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls

General Terms

Security

Keywords

attribute based access control, constrained logic programming with sets

1. INTRODUCTION

Open environments such as the Internet where service requesters are not identified by unique names depend upon their attributes (usually substantiated by certificates) to gain accesses to resources. In order to accommodate this need, many important attribute based access control systems have been designed in the recent past [LMW02b, BS00, BS02, YWS00, YWS01, YWS03]. Also role based [SCFY96, BS04] and flexible [JSSS01, BCFP03] access control systems can be used to specify some aspects of attribute based access policies by exploiting the indirection and the collectability of permissions provided by roles. One of the important aspect of attribute based access control policies is their ability to specify accesses to a collection of services based upon a collection of attributes processed by the requester. Thus the nature of such *collections* and their properties determines the expressibility of specifiable policies. Some systems such as [LMW02b] model these collections as sets (but with limited structural properties) and others as finite vectors of attributes [BS00, BS02]. Yet others use roles as their primary vehicle of *collecting* attributes and services. To the best of our knowledge, there is no single model that uses *sets* as data structures with their algebraic operations (i.e. $\cup, \cap, \setminus$) to specify attribute based policies. This paper does so by using a version of computable set theory as a constraint system in logic programming.

The version of set theory we use is CLP($\mathcal{SET}$), the *hereditarily finite* and computable set theory developed by Dovier et al. [DPPR00, DPR00, DPR98]. Hereditarily finiteness refers to the fact that sets are constructed out of a finite universe by applying operators such as $\cup, \cap$ etc. Because our policies refer to attributes and services, we use a two sorted first order language with set variables. The chosen constraint system ensures that set terms satisfy an equality theory with algebraic identities such as the distributivity of unions over intersections etc. As will be seen shortly, because policies are written as stratified constraint logic programs with recursion, they terminate as logic programs. Also the specification language follows the blueprint of the *flexible access framework (FAF)* [JSSS01], where conflict resolution and default policies are specifiable - thereby ensuring the consistency and completeness of policies.

Fixed point semantics of constraint logic programs assigns one of three truth values *true*, *false* and *undetermined* to every predicate instance. This is unacceptable for an access controller because every access request requires a unique *yes* or *no* answer. But we show that the policies we allow are always assigned either *true* or *false*. Additionally, constraint

*This work was partially supported by the National Science Foundation under grant CCR-0113515.
†Thanks to William H. Winsborough for valuable comments

FMSE'04, October 29, 2004, Washington, DC, USA.

logic programs that we use as policies have a NP complete run time, that invite concerns about their utility by access controllers. As a remedy, we show that our policies can be rewritten to yield faster runtimes by applying an appropriate set of unfolding transformations that have the same semantics and runtime advantages as materialized views.

The rest of the paper is organized as follows. Section 2 describes related work. Sections 3 and 4 provides the syntax and semantics of our language. Section 5 describes materialization and policy transformation for execution time efficiency. Section 6 concludes the paper.

2. RELATED WORK

The *RT* framework of Li et al. [LMW02a] is a distributed, identity-less access control specification framework where each role specifies the roles that it contains and/or attributes that are required for membership. They use the predicate `isMemeber(x,X)` to model that `x` is a member of the role `X`. Although the RT syntax does not explicitly support set operations such as $\cup, \cap, \subset, \setminus$, they have a notion of *intersection roles* to specify those attributes that are contained in other roles defined using attribute sets. In contrast, having set operations allows our policies to express set unions and intersections in a more intuitive syntax satisfying structural identities expected of sets. Secondly, RT uses only Horn clauses, thereby preventing the use of set difference operator, consequently preventing from constructing *difference roles*, whereas we admit limited forms of negations in rule bodies and allow the set difference operator. Furthermore RT is based on a monotonicity assumption where any superset of a set of attributes automatically satisfy the requirements specified by the set, which we do not require. Nevertheless, RT addresses trust propagation and distributing access specifications that we do not have.

The work of Bonatti et al. [BS00, BS02] is another identity-less access control system where credentials and services are modeled as vectors of attributes. They support credential and services hierarchies. Although the framework uses vectors, their policies do not seemed to exploit attribute ordering. Thus, to the best of our understanding, vectors in [BS00, BS02] behave like collections, that we model as sets. In addition, policies of [BS00, BS02] are arranged in a three level hierarchy. Using recursion, our framework can have a hierarchy with many levels.

Yu et al. [YWS00, YWS01, YWS03] developed a service negotiation framework for requesters and providers to gradually expose their attributes. Fully instantiated attribute sets are traded in [YWS03], extending [YWS01] to support credentials with internal structure, although the problem is not completely cast and resolved in logic.

Although we use the design blueprints of FAF [JSSS01], it does not have set variables. Thus, this work can be considered as an enhancement of FAF to the domain of attribute based policies.

3. SYNTAX

As stated, we use two sorts of sets to model attributes and services in CLP($\mathcal{SET}$) [DPPR00, DPR00, DPR98]. The constraint logic programming language we use to formalize attribute based access control consists of terms constructed the usual way from variables and functions. We also have two kinds of predicates - those used to specify the computation domain and those used to specify its sub domain of constraints. In addition to satisfying the usual boolean algebraic laws such as associativity, commutativity etc, one aspect of the hereditarily finiteness of CLP($\mathcal{SET}$) is that, these sets satisfy the axiom of foundation in Zermelo-Fraenkel (ZF) set theory [Kun80], which we exploit to ensure the termination of all queries.

3.1 The Nature of Sets

Following Dovier et al. [DPPR00], our language consists of four sorts, two for attributes and two for services. They are given as Ker_a, Set_a, Ker_s and Set_s. Ker_a and Ker_s are the basic sorts for attributes and services. Set_a and Set_s are for hereditarily finite sets constructed over Ker_a and Ker_s respectively. Each sort has its own constants and function symbols. We assume that Ker_a and Ker_s has two constant symbols $\perp_a$ and $\perp_s$. These are useful in modelling partial functions as our application domain requires them. When clear from the context we drop the subscript and use $\perp$ for brevity. We assume that Set_a and Set_s has constants $\emptyset_a$ and $\emptyset_s$ respectively to denote the null sets of their respective sorts. For brevity we drop the subscripts and use $\emptyset$ for both. To create sets of attributes and services we have two binary function symbols $\{- \mid -\}_a$ and $\{- \mid -\}_s$. Their sorts are $(\{Ker_a, Set_a\} \uplus \{Ker_a\}) \mapsto \{Set_a\}$ and $(\{Ker_s, Set_s\} \uplus \{Ker_s\}) \mapsto \{Set_s\}$ respectively. For brevity when the subscript is clear from the context, we drop it and use $\{- \mid -\}$ for $\{- \mid -\}_a$ and $\{- \mid -\}_s$. Intuitively, $\{a|X\}$ represents the set $\{a\} \cup X$. For brevity we also use $\{a\}$ to represent $\{a|\emptyset\}$.

EXAMPLE 1. *Suppose a digital library provides services for checking membership status, browsing and printing. Browsing is specialized to the table of contents (ToC), abstracts and file contents. Printing specializes to printing on* `letter` *size and* `A4` *paper. Thus the service hierarchy has* `digital library service (dlS)` *as its root with three children* `br`*, (for browse)* `ckStat` *and* `print`*. Furthermore, the* `print` *service has two children* `letter` *and* `A4`*, and the* `browsing` *service consists of* `brToC`*,* `brAbs` *and* `brCont` *as children. Accordingly, in this example, the service hierarchy is represented as the set of path names as {*{*dlS*}*,*{*dls,br*}*,*{*dlS,ckStat*}*, {*dls,print*}*, {*dls,br,brToC*}*, {*dls,br,brAbs*}*,*{*dls,br, brCount*}*}. If there is likely to be any confusion about having the same name repeated in the hierarchy, then we can avoid that by using representation such as {*parent*, {*$child_1$*}, {*$child_2$*}, ..., {*$child_n$*}}*

Firstly, these sets of sets ... sets of elements are the hereditarily finite. Furthermore non-set elements are chosen from a countable alphabet, although any single set constructed will have only finitely many of them. Thus, semantically our language of sets in CLP($\mathcal{SET}$) constructs hereditarily finite sets over an uninterpreted Herbrand base.

*Secondly, attributes of the requester (usually conveyed by submitting credentials or certificates) can also be represented as nested sets. We have two disjoint attribute hierarchies, one for membership status and the other for payments. The membership hierarchy is modeled by {*patron*, {*patron,member*}*,*{*patron,senior*}*, {*patron, fellow*}*}. The payment hierarchy is modelled by {*payment*, {*payment,dollar*}*,*{*payment,euro*}*}*

Thirdly, Set_a or Set_s, can be used to create sets using $\emptyset$, such as $\{\emptyset, \{\emptyset\}\}$. As will be seen shortly, we use such

nested sets to limit the recursive backtracking through rule chains. We also use nested sets to code integers. For example $\{\{\ldots\{\emptyset\}\ldots\}\}$ *where the empty set* $\emptyset$ *is embedded in* $\mathtt{n}$ *braces is used to represent the integer* $\mathtt{n}$. ■

Following Dovier et al. [DPPR00], we take $\{=, \neq, \in, \notin, \cup_3, \not\cup_3, \|, \not\|\}$ as our constraint predicates for each set sort - namely Set_a and Set_s. In addition, $\subseteq, \cap_3, \setminus$ and their negations can be defined using the former, making all of them available as constraint predicates. Here $\cup_3$ is the ternary predicate $X \cup_3 Y = Z$, An analogous explanation applies for $\cap_3$. Similarly $X \parallel Y$ holds iff $X \cap Y = \emptyset$. Our constraints are conjunctions and disjunctions of constraint predicates. In addition, we consider the following collection of (reserved) predicate symbols that relate terms of Set_a and Set_s.

- `cando`(X,Y,±,Z) is a 4-ary predicate where X and Y are attribute and service set terms. The third attribute is either + or − (We can use $\emptyset$ and $\{\emptyset\}$ to encode + and - as sets). The fourth variable Z is a set term used to encode the recursive depth. The intuitive reading of `cando`(X,Y,±,Z) is that a holder of the attribute set X is authorized/prohibited in using to services Y depending on the sign + or -. The nesting of the $\emptyset$ says the recursive depth. That is `cando`({adult,liveInVA}, {PG13,X},+,{$\emptyset$}) says that any holder of attributes {adult,liveInVA} is entitled to the services {PG13, X}, and this is stated as a fact derivable with one level of backtracking.
- `dercando`(X,Y,±,Z) is a 4-ary predicate with the same set of parameters as `cando`. The only differnce between `cando`and `dercando`is that the latter can be used in recursive rules.
- `do`(X,Y,+,Z) is a 4-ary predicate with the same set of parameters as `cando`. `do`(X,Y,+,Z) expresses a final authorization to permit/prohibit a holder of the attribute set X is in using to services Y depending on the sign + or -.

Definition 1 (ABAC rules and policies). *ABAC policies are constructed using reserved predicates and possibly other application specific predicates as follows.*

1. *Rules using* `cando` *heads must be of the form* `cando`$(X, Y, \pm, \{\emptyset\}) \leftarrow B$ *where the body* B *must not have any other reserved predicates. These are used to state basic facts about granting/denying access to services.*
2. *Rules using* `dercando` *heads must conform to the following restrictions.*
   (a) `dercando` *can appear in their bodies only positively.*
   (b) *The bodies of a rule with a* `dercando` *head can have* `cando` *and non-reserved predicates.*
   (c) *Any rule with a* `dercando` *head must be of the form given below where* `dercando`$(-, Y-, \pm, Z_1)$, …, `dercando`$(-, Y-, \pm, Z_n)$ *are the only occurrences of* `dercando` *in the body and* L_i *are either* `cando` *or any application specific (non-recursive) predicate. Here the - in predicate instances such as* `dercando`$(-, Y-, \pm, Z_1)$ *means that the term could be anything of the appropriate sort.*

$$\begin{aligned}\texttt{dercando}(-,-,-,Z) \;\leftarrow\; & \texttt{dercando}(-,Y-,\pm,Z_1),\ldots \\ & \texttt{dercando}(-,Y-,\pm,Z_n), \\ & Z_1 \in Z, \ldots Z_n \in Z \\ & L_1,\ldots L_m,\ldots,Z_1 \in Z,\ldots Z_n \in Z\end{aligned} \tag{1}$$

3. *Rules with a* `do`$(X, Y, +, Z)$*, as their head must also conform to the second restriction for* `dercando` *but can have only* `dercando`*,* `cando` *or application specific (non-recursive) predicates in their body.*
4. *The only rule with a* `do`$(-,-,-,-)$ *head is of the form* `do`$(X, Y, -, \{Z\}) \leftarrow \neg$`do`$(X, Y, +, Z)$*, where the third attribute is a negative sign.*

Any finite collection of rules conforming to constraints (1) through (3) and one rule (4) is said to be an ABAC policy. We usually use $\mathcal{P}$ *as a symbol for an ABAC policy.*

Example 2. *A policy for the digital library hierarchies in example 1 is that members can check their* membership status *by submitting the membership ID or alternatively using their name and mother's maiden name. Any member can also browse the table of contents (toc). Senior members and fellows are allowed to browse the abstracts and contents. Printing is free for fellows, but others pay for printing privileges. Members paying in* `dollars` *print on* `letter` *quality paper and those paying in* `euros` *print on* `A4` *paper.*

$$\texttt{cando}(\{y\}, X, +, \{\emptyset\}) \;\leftarrow\; memID(y), memStatus(\{y\}, X). \tag{2}$$

$$\texttt{cando}(\{y, z, \{y\}\}, X, +, \{\emptyset\}) \;\leftarrow\; isAName(y), isAName(z), memMother(y, z), memStatus(\{y, z\}, X) \tag{3}$$

$$\texttt{dercando}(X, Y, Z, \{U\}) \;\leftarrow\; \texttt{cando}(X, Y, Z, U) \tag{4}$$

(5)

$$\texttt{dercando}(U, \{\{dlS, br, brTOC\} \mid X\}, +, \{Z\}) \;\leftarrow\; \texttt{dercando}(U, X, +, Z), X \neq \emptyset \tag{6}$$

We have an application specific binary predicate $memStatus(-,-)$*, with two* set *arguments, where* $memStatus(X, Y)$ *holds iff* Y *is the profile of the entity identified by the attribute set* X*. For example, a user identified by the member ID where* $X = \{ID_1234\}$ *has the profile* $Y = \{senior-member, began-01-01-1960, paidTo-05-05-2005, Address-Modena-Italy\}$*. We use three other application specific predicates* $memID(-)$*,* $isAName(-)$*, and* $memMother(-,-)$*.* $memID(x)$ *is true if* x *is a member, and* $isAName(x)$ *holds iff* x *is a name. Appropriate instances of* $memMother(x, y)$ *must exists at the access controller.*

Rule (3) says that X *is the membership status that can be obtained for the attribute* $\{y\}$ *(i.e. member ID). Rule (4) says that* X *is the obtainable membership status for the attributes* $\{name, mothersname\}$ *pair. (Notice that* {y,z,{y}} *is used to model the ordered pair (y,z) as a set in ZF set*

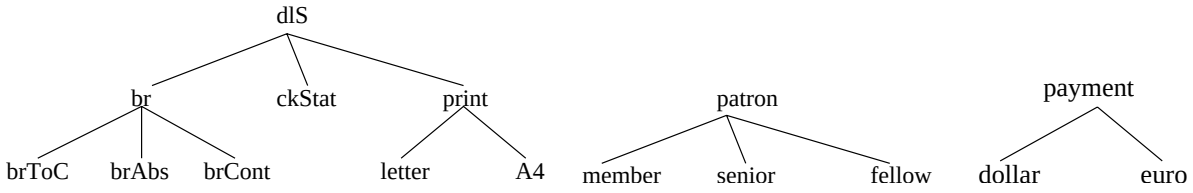

Figure 1: An example of Attribute Hierarchies

$$\texttt{dercando}(U, \{\{dlS, br, brTOC\}, \{dlS, br, brAbs\}, \{dlS, br, brCont\} \mid X\}, +, \{Z\}) \leftarrow senior \in X, \texttt{dercando}(U, X, +, Z) \quad (7)$$

$$\texttt{dercando}(U, \{\{dlS, br, brTOC\}, \{dlS, br, brAbs\}, \{dlS, br, brCont\} \mid X\}, +, \{Z\}) \leftarrow fellow \in X, \texttt{dercando}(U, X, +, Z) \quad (8)$$

$$\texttt{dercando}(\{pay, \{pay, dollar\} \mid U\}, \{print, \{print, letter\} \mid X\}, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), fellow \notin X, X \neq \emptyset, Z' \in Z \quad (9)$$

$$\texttt{dercando}(\{pay, \{pay, euro\} \mid U\}, \{print, \{print, A4\} \mid X\}, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), fellow \notin X, X \neq \emptyset, Z' \in Z \quad (10)$$

$$\texttt{dercando}(U, \{print \mid X\}, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), fellow \in X, Z' \in Z \quad (11)$$

$$\texttt{do}(U, X, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), Z' \in Z \quad (12)$$

$$\texttt{do}(U, X, -, Z) \leftarrow \neg\texttt{do}(U, X, +, Z'), Z' \in Z \quad (13)$$

theory.) Notice that the membership status obtainable using rules (3)and (4) may not be the same, as we are in an identity-less system. As seen, specified policies do not have to divulge the same user profiles. Rule (5) facilitates using information available in the `cando` *predicates to be used in recursive queries. Rule (7) permits any member (identified by having a nonempty profile) to browse the table of contents of the digital library. Similarly, rules (7) and (8) state that senior members and fellows can browse abstracts and the contents of the digital library. Rule (9) and (10) states that in addition to other privileges, non-fellows can either pay in dollars and obtain prints on letter paper or pay in euros and obtain copies on A4 paper. Rule (11) says that fellows can print, without paying. Conversely, the policy does not state what paper must be used by them. If we wanted to allow both kinds of paper, either inclusively or exclusively then the rule (11) could be modified to rule (14) or rules (15) and (16). Finally rules (12) and (13) applies the policy of prohibiting any accesses that are not explicitly permitted by the previous rules. Notice that the given policy explicitly stratifies all instances of rules, where the strata are given by the rank of the last variable of the head predicates. All* `cando` *predicates belong to the first strata as the rank of its first predicate* $\{\emptyset\}$ *is 1. All instances of* `dercando` *predicates where the service term has browsing options have strata 3, as the* `dercando` *head in rule (5) has strata 2, and therefore any* `dercando` *instance from rules (7) through (8) has strata 3. Similarly, any* `dercando` *predicate where the service term has a printing term has strata 4, as these come from rules (9) through (11). Finally, according to rules (12) and (13), all instances of* `do` *with a (+) third variable instance has rank 5, and those with a (-) third variable instance has rank 6. But again, rules 7 through 11 can be recursive, giving many other ranks for these predicates instances.*

4. SEMANTICS

This section describes models of ABAC policies. As stated in Definition 1, an ABAC policy consists of a finite collection of rules with `cando`, `dercando` and `do`(-,-,+,-) heads and one rule with a `do`(-,-,(-),-) head. Of these rules, only `dercando` rules are recursive. But as a constraint logic program, it has a three valued Kripke-Kleene model [Kun87, Fit85] where every predicate instance evaluates to one of three truth values *true, false* or *undefined*. We will shortly show that every query (a request) will evaluate to either *true* or *false*, and therefore has only two truth values - ensuring that every access request is either granted or denied. Because we allow nested negative predicates, we need to interpret *negation*. We can either use negation as failure or *constructive* negation [Cha88, Cha89] as proposed by Fages [FG96, Fag97]. This is because the third alternative namely using constructive negation as proposed by Stuckey [Stu91, Stu95] requires that the constraint domain be *admissibly closed*. But Dovier shows that set constraints as we use them in ABAC policies are not admissibly closed, and proposes an alternative formulation to handle nested negations [DPR01]. Conversely, at the cost of requiring some uniformity in computing negated subgoals of a computation tree, Fages's formulation does not require the constraint domain to be admissibly closed [FG96, Fag97]. Formalities follow. We first repeat some standard definitions in [Fit02] to clarify notations. This enables us to describe a materilization structure for three-valued models in section 5.2.

$$\texttt{dercando}(U, \{print, \{print, letter\}, \{print, A4\} \mid X\}, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), \quad (14)$$
$$fellow \in X, Z' \in Z$$

$$\texttt{dercando}(U, \{print, \{print, letter\} \mid X\}, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), \quad (15)$$
$$fellow \in X, Z' \in Z$$

$$\texttt{dercando}(U, \{print, \{print, A4\} \mid X\}, +, Z) \leftarrow \texttt{dercando}(U, X, +, Z'), \quad (16)$$
$$fellow \in X, Z' \in Z$$

DEFINITION 2 (P^*, T_P AND $\Phi_P \uparrow$ OPERATORS). *Let P be a logic program, and let P^* be all ground instances of clauses in P. We take $A\leftarrow$ as A and any ground atom A not in the head of any rule as $A\leftarrow$ **false**. We now define two and three valued truth lattices to be $\mathbf{2} = \langle\{T, F\}, <_2\rangle$ and $\mathbf{3} = \langle\{T, F, \bot\}, <_3\rangle$ respectively, where T, F and $\bot$ are taken to mean **true**, **false** and **unknown** truth values. Partial orderings $<_\mathbf{2}$ and $<_\mathbf{3}$ satisfy as $F <_2 T$ and $\bot <_3 T, \bot <_3 F$ respectively. A mapping V from ground atoms of P to $\mathbf{2}$ and $\mathbf{3}$ is said to be respectively a two-valued or a three-valued valuation of P. Given a valuation V, the two and three valued immediate consequence operators $T_P(V)$ and $\Phi_P(V)$ are defined as follows.*

$T_P(V)$: $T_P(V) = W$ is defined as:

- *$W(H) = T$ if there is a ground clause $H\leftarrow B_1, \ldots, B_n$ in P^* such that $V(B_i) = T$ for $i \leq n$.*
- *$W(H) = F$ otherwise.*

$\Phi_P(V)$: $\Phi_P(V) = W$ is defined as:

- *$W(H) = T$ if there is a ground clause $H\leftarrow B_1, \ldots, B_n$ in P^* such that $V(B_i) = T$ for $i \leq n$.*
- *$W(H) = F$ if for every ground clause $H\leftarrow B_1, \ldots, B_n$ in P^* where $V(B_i) = F$ holds for some $i \leq n$.*
- *$W(H) = \bot$ otherwise.*

In evaluating Φ, negation is interpreted as $\neg T = F, \neg F = T$ and $\neg\bot = \bot$. Now we define bottom-up semantic operators for both T_P and Φ_P, where Ψ stand for either of them in the following.

- *$\Psi^0 \uparrow (P) = V_{false}$, where V_{false} assigns F (false) to all instantiated atoms.*
- *$\Psi^{\alpha+1} \uparrow (P) = \Psi(\Psi^\alpha \uparrow (P))$ for every successor ordinal α.*
- *$\Psi^\alpha \uparrow (P) = \bigvee_{\beta<\alpha}(\Psi^\beta \uparrow (P))$ for every limit ordinal α.*

For Horn clauses (i.e. those without negative non-constraint predicates in the body) $T(P)$ has a least fixed point $T_\omega(P)$, which is considered the model of P [Fit02]. Nevertheless, as shown in [Fit02], for three-valued semantics, the least fixed point may not be obtained at ordinal ω. But following standard practice we take $\Phi_\omega(P)$ as the *meaning* (i.e. semantics) of an ABAC policy $\mathcal{P}$ as formalized in definition 3.

DEFINITION 3 (BOTTOM-UP SEMANTICS). *Let $\mathcal{P}$ be an ABAC policy and Φ be the three-valued immediate consequence operator stated in definition 2. Then we say that $\bigvee_{i\in\omega} \Phi^i(\mathcal{P})$ is the model of $\mathcal{P}$.*

Definition 3 says that we obtain a model of $\mathcal{P}$ by evaluating the Φ operator ω many times. As promised, we now show that $\bigvee_{i\in\omega} \Phi^i(\mathcal{P})$ only takes two truth values. In order to do so, we consider a version of the standard operational semantics for constraint logic programs. Thereafter by defining a *rank* for a formula so that the rank decreases as one proceeds from the root towards the leaves of a top down computation tree, we show that every computation terminates. The property we use here is the well-foundedness of the membership predicate $\in$ built into the fourth variable of `cando`, `dercando` and `do` predicates. In order to do so, we now repeat (a version of) operational semantics proposed for constraint logic programs [JL87, Koz98].

DEFINITION 4 (OPERATIONAL SEMANTICS). *A state is a pair (A, C) of multisets of predicates A and constraints C. Let $\mathcal{P}$ be an ABAC policy and (A, C) (A', C') be states. We say that:*

- *$(A \cup \{p(\vec{s})\}, C) \rightarrow_1 (A \cup B, C \cup C" \cup \{\vec{s} = \vec{t}\})$ is a one-step derivation provided $p(\vec{t})\leftarrow B, C"$ is a renamed apart instance of a rule in $\mathcal{P}$.*
- *We say that (A, C) fails if $A \neq \emptyset$ and there is no predicate $p \in A$ where $p(\vec{t})\leftarrow B, C"$ is a rule in $\mathcal{P}$.*
- *We say that (A, C) is successful if $(A, C) \rightarrow_* (\emptyset, C')$ for some constraint set C' satisfiable by an assignment σ of variables to values, where $\rightarrow_*$ is the reflexive transitive closure of $\rightarrow_1$.*
- *A query (A, C) is said to flounder if it neither fails nor is successful.*

The third clause of definition 4 usually reads as (A, C) is said to be successful if $(A, C) \rightarrow_* (\emptyset, C')$ for some consistent constraint set C'. But Dovier et al. shows that in the computable set theory we use, a set of constraints C' is consistent iff it is satisfiable by some assignment of variables to values [DPPR00, DPR00, DPR98]. Coincidentally, the operational semantics given by definition 4 and the fixed point semantics given by definition 3 coincide [JL87, Koz98]. We now proceed to show that ABAC policies do not flounder.

DEFINITION 5 (RANKS). *We say that the rank of literals with a ground fourth attribute is the maximum nesting of braces in it, formally defined as:*

$$rank(s) = \begin{cases} max\{1 + rank(u), rank(v)\} \\ \qquad \text{if } s \text{ is } \{u \mid v\} \\ 0 \\ \qquad \text{if } s \text{ is not of the form } \{u \mid v\} \end{cases}$$

We say that the rank of a reserved predicate where its fourth attribute is ground is the rank of its fourth attribute, and the rank of a ABAC rule with a ground fourth attribute is the rank of its head predicate.

Suppose A is a finite multiset of ABAC literals where the fourth attributes are ground, and $\{a_1, \ldots, a_n\}$ lists elements of A in the decreasing order of their ranks. That is, they satisfy the condition that $i > j \rightarrow R(a_i) \geq R(a_j)$, where $R(a_j)$ is the rank of a_j. Suppose m is the largest rank in A. That is, $m = max\{R(a) : a \in A\}$, and let $\mathcal{C}(A,i) = \mid \{R(a_j) : R(a_j) = i\} \mid$ for every $i \leq m$. That is, $\mathcal{C}(A,i)$ is the number of predicates with rank i. Then define $R(A)$, the rank of the multiset A as the vector $(\mathcal{C}(A,m), \ldots, \mathcal{C}(A,0))$. We order multi set ranks (that is $(\mathcal{C}(A,m), \ldots, \mathcal{C}(A,0))$) lexicographically.

Definition 5 specify the ranks for ground instances of reserved predicates, their multisets and ABAC rules. Using our operational semantics, we show that any application of an ABAC rule reduces the rank of the *rule state*, and therefore must terminate finitely.

LEMMA 1 (PROPERTIES OF RANKS). *Suppose $h \leftarrow B$ is an ABAC rule with a ground fourth attribute. Then $R(h) > R(b)$ for any reserved predicate b in the body B. Furthermore, suppose that $(A \cup \{p(\vec{s})\}, C) \rightarrow_1 (A \cup B, C \cup C" \cup \{\vec{s} = \vec{t}\})$ is a one-step derivation where $p(\vec{t}) \leftarrow B, C"$ is a rule in $\mathcal{P}$ and $p(\vec{s}) \leftarrow B, C"$ is a named apart instance of $p(\vec{s}) \leftarrow B, C"$. Then $R(A \cup \{p(\vec{s})\}) > R((A \cup B)$. Here $p(\vec{s})$ and $p(\vec{t})$ must have ground fourth attributes.*

Proof: See the appendix. We now use lemma 1 to show that ABAC queries terminate.

THEOREM 1 (FINITE TERMINATION OF ABAC QUERIES). *Every ABAC query (A, C) either fails or succeeds, where A is a reserved predicate with a ground fourth attribute.*

Proof: See the appendix.

As a corollary, we now obtain that any ABAC query always gives a *yes* or *no* answer, implying that all three valued models have only two truth values *true* and *false*, as stated in the follwing corollary.

COROLLARY 1. *Every three valued model of a ABAC policy assigns either T or F for reserved predicates where the fourth attribute is instantiated. In that case, bottom-up semantics and the well-founded constructions assigns the same truth values to the same predicate instances and have the same answer sets.*

Proof: See [BS04]. ■

Corollary 1 shows that in ABAC every request is either honored or rejected. But the ABAC model is not a fixed point of the Φ operator , as it is well known that the closure ordinal of the Φ operator is not ω [FBJ90, Fag97]. ([FBJ90] gives a simple counter example)

5. OPTIMIZING ABAC POLICIES

One of the major criticisms levied against using (constrained) logic programs is their *runtime inefficiency* due to the backtracking through program clauses. Although general complexity bounds arising out of constraint solvers cannot be totally avoided, we choose two techniques (among many available techniques such as stack copying, constraint optimization etc) to reduce this inefficiency. The first is to transform any ABAC policy into one with lesser backtracking but the same semantics - generally referred to as program transformations. The second is to materialize commonly accessed predicates instances. We discuss them in order, and show that they provide the same level of efficiency.

5.1 Applying Program Transformation Techniques to ABAC Policies

As stated, the objective is to transform an ABAC policy into one that is semantically equivalent policy, but with lesser runtime overheads. General techniques of this kind grew out of program transformation work for functional languages by Burstall and Darlington [BD77], and were later applied to logic programming by Tamaki and Sato [TS84]. They are comprehensively surveyed by Petrossi et al. [PP98]. Etalle et al. [EG96], Maher [Mah93] and some reverences quoted therein have extended these results to constraint logic programming. In this section we develop a policy rewriting algorithm based on the work of Etalle et al. [EG96] for a restricted class of policies that use only Horn clauses. The reason for this limitation is imposed upon the theorems proved in [EG96] that we use as the basis of our algorithm. Our ongoing work address extending them to constructive negation ala Fagas. First we state results used from [EG96] in our algorithm. Toward that end, definition 6 specify the program transformations we propose to apply to ABAC policies.

DEFINITION 6 (CLP TRANSFORMS). *Suppose $\texttt{cl}$ is the rule $A \leftarrow C, H, K$ in the ABAC policy $\mathcal{P}$ where H and K are respectively a non-constraint predicate and a sequence of them. Then:*

Unfolding: *Let $\texttt{cl}$ be the rule $A \leftarrow C, H, K$ and $\texttt{cl}_i$ be $H_i \leftarrow C_i, B_i$ for all $i \leq n$. Let $\{\texttt{cl}_i \mid 1 \leq n\}$ be all rules in $\mathcal{P}$ where $C \wedge C_i \wedge (H = H_i)$ is satisfiable. Let $\texttt{cl}'_i$ be the clause $A \leftarrow C \wedge C_i \wedge (H = H_i), B_i$. Then* unfolding *$H$ in $\mathcal{P}$ consists of replacing $\texttt{cl}$ by the collection $\{\texttt{cl}_i : 1 \leq i \leq n\}$ to obtain $\mathcal{P}'$. That is, $\mathcal{P}' = \mathcal{P} \setminus \{\texttt{cl}\} \cup \{\texttt{cl}_i : 1 \leq i \leq n\}$.*

Clause Splitting: *Let $\texttt{cl}_i$ be the rule $H_i \leftarrow C_i, B_i$ where $\{H_i \leftarrow C_i, B_i \mid 1 \leq i \leq n\}$ are all clauses in $\mathcal{P}$ such that every $c \wedge C_i \wedge (H = H_i)$ is satisfiable. Let $\texttt{cl}'_i$ be the rule $A \leftarrow C \wedge C_i \wedge (H = H_i), H, K$ for all $i \leq n$. If for any $i, j \in [1, n]$, $C \wedge C_i \wedge C_j \wedge (H_i = H_j)$ is inconsistent, then replace $\texttt{cl}$ with the collection $\{\texttt{cl}_i : 1 \leq i \leq n\}$ to obtain $\mathcal{P}'$. That is, $\mathcal{P}' = \mathcal{P} \setminus \{\texttt{cl}\} \cup \{\texttt{cl}_i : 1 \leq i \leq n\}$. Accordingly, splitting is an unfolding in which bodies of the unfolding clauses are not replaced.*

Clause Removal: *Let $\texttt{cl}$ be the cluse $H \leftarrow C, B$ in $\mathcal{P}$ where C is unsatisfiable. Then remove $\texttt{cl}$ from $\mathcal{P}$ to obtain $\mathcal{P}'$. That is, $\mathcal{P}' = \mathcal{P} \setminus \{\texttt{cl}\}$.*

Constraint Replacement: *Suppose C' is a constraint where every successful derivation of $B \rightarrow_* D$ satisfies $(C \wedge D) \leftrightarrow (C' \wedge D)$. Then replace $H \leftarrow C, B$ by $H \leftarrow C', B$.*

Theorem 2 from [EG96] says that program transformations in definition 6 do not alter the semantics of ABAC policies.

THEOREM 2. *Suppose $\mathcal{P}'$ is obtained from $\mathcal{P}$ by applying any finite sequence of program transformations stated in definition 6. Then $\mathcal{P}$ and $\mathcal{P}'$ have the same answer set semantics.*

Proof: See [EG96]. ■

Algorithm 1 policy transformation algorithm

INPUT : An ABAC Policy $\mathcal{P}$
OUTPUT : An ABAC Policy $\mathcal{P}'$ with the same semantics as $\mathcal{P}$

Loop for predetermined number of times for rules without `dercando`(-,-,(-),-) heads
if rule $H \leftarrow C, B \in \mathcal{P}$ **then**
  %%Comment: remove rules with unsatisfiable constraints.
  Apply constraint solver to C
  if C is inconsistent **then** remove $\{H \leftarrow C, B\}$ from $\mathcal{P}$
  %%Comment: reduce constraints
  else let C' be the reduced constraint of C. Replace $H \leftarrow C, B$ by $H \leftarrow C'B$.
else if for rule $A \leftarrow C, H, K$ where $\{H_i \leftarrow C_i, B_i \mid 1 \leq i \leq n\}$ are the only clauses in P and $c \wedge C_i \wedge (H = H_i)$ is satisfiable for all $i \leq n$ **then**
  if for any $i, j \in [1, n]$, $C_i \wedge C_j \wedge (H_i = H_j)$ is inconsistent **then**
  %% Comment: apply splitting
  replace $A \leftarrow C, H, K$ by the rule set $\{A \leftarrow C \wedge C_i \wedge (H = H_i), H, K\}$.
  else *%%Comment: apply unfolding*
  replace $A \leftarrow C, H, K$ by the set of rules $\{A \leftarrow C_i \wedge C \wedge (H = H_i), B_i \mid 1 \leq i \leq n\}$.
end if

We now use transformations specified in definition 6 to create rules at policy analysis time (i.e. compile time) that reduce the backtracking overhead incurred at policy application time (i.e. runtime). Notice that in algorithm 1, we only consider rules without `do`(-,-,(-),-) heads and repeatedly apply program transformations that provably preserve correct answer set semantics. Thus the algorithm is correct by a simple application of theorem 2. Now we show how this algorithm can reduce the runtime cost for policies for example 2. Notice rules other than the last rule with a `do`(-,-,(-),-) head in example 2, are Horn clauses and therefore algorithm 1 apply to them.

EXAMPLE 3. *The following rules are obtained by applying algorithm 1 to rules given in example 2.*

Applying unfolding to rules (3) and (5) derive rule (17). Similarly, unfolding rules (4) and (5) results in rule (18). Similarly, unfolding rules (17) and (7) results in rule (19). Unfolding rule (19) and (12) results in rule (20). Directly executing rule (20) does not require a backtracking algorithm to be executed at runtime, although it require evaluating the same basic predicates.

Notice that applying the stated sequence of unfolding transformations leave rules (17) through (20) as the new ABAC policy. Similarly, by unfolding other rules, we end up with a policy where all defined predicate other than `do`(-,-,-,(-),-), have basic predicates in their body. Such a representation can be considered a *canonical* representation for ABAC policies. The final step of unfolding them against valid instances of base predicates to this canonical form will reduce all rules to valid instances of predicates. This is shown in section 5.3.

5.2 Materializing ABAC Policies

As a secondary optimization of runtime costs, we propose to materialize ABAC policies. Because ABAC policies are locally stratified , our (soon to be described) *materialization structure* is recursively built using the stratification order. We use an approach similar to that used in [JSSS01] to build a materialization structure, but appropriately altering it to suit ABAC policies. Towards this end, we first (re)-define the materialization structure differently (from [JSSS01]) and accordingly its corresponding notion of correctness with respect to ABAC policies.

DEFINITION 7 (MATERIALIZATION STRUCTURE). *A materialization structure $\mathtt{MS}(\mathcal{P})$ for an ABAC policy $\mathcal{P}$ is a set of pairs (A, I), where A is a ground atom and I is a set of (indices of) rules of the form $H \leftarrow C, B$. $\mathtt{MS}(\mathcal{P})$ is said to* correctly model *$\mathcal{P}$ iff the following conditions hold.*

1. *$\Phi_\omega \uparrow (\mathcal{P})(H(\vec{c})) = T$ iff there is at least one pair $(H(\vec{c}), I) \in \mathtt{MS}(\mathcal{P})$ for some index set I satisfying $\hat{\mathtt{cl}} \in I$ for each rule $\mathtt{cl}$ of the form $H \leftarrow C, B$ where C is the constraint part and B is the non-constraint part of the rule body.*
2. *Suppose $\Phi_\omega \uparrow (\mathcal{P})(B(\vec{c}, \vec{c'})) = T$ for a rule $\mathtt{cl}$ as stated in (1) where $\vec{c}$ are all the instantiations for variables of H and $\vec{c'}$ are all the extra constants required to fully instantiate other variables of B. (Notice that the body can have more variables than the head of a rule) If $C(\vec{c}, \vec{c'})$ is valid then there is an index I such that $\hat{cl} \in I$ and $(H(\vec{c}), I) \in \mathtt{MS}$.*

According to definition 7 a materialization structure correctly models a policy $\mathcal{P}$ iff every instance of an atom A that is true in the Kripke-Kleene closure contains a pair (A, I) where I is a set of (index of) rules that directly support the truth of A. Given a materialization structure $\mathtt{MS}(\mathcal{P})$ of a policy $\mathcal{P}$ the model $\Phi_\omega \uparrow (\mathcal{P})$ of $\mathcal{P}$ is then the projection over the first element of the pairs that are evaluated to be true by $\Phi_\omega \uparrow (\mathcal{P})$. The materialization structure and the Kripke-Kleene model at stratum i are denoted by $\mathtt{MS}_i(\mathcal{P})$ and $\Phi_i \uparrow (\mathcal{P})$ respectively. Algorithm 2 uses the step-wise construction of the Kripke-Kleene model to produce the materialization structure of an ABAC policy. In order to present the algorithm, we need the following technical definition about *adding* entries into a materialization structures.

DEFINITION 8 ($\oplus$). *Let $\mathtt{MS}(\mathcal{P})$ be a materialization structure, A a ground instance of a non-constraint literal and S a set of rules.*

$$\mathtt{MS}(\mathcal{P}) \oplus (A, \hat{\mathtt{cl}}) = \begin{cases} \mathtt{MS}(\mathcal{P}) \setminus \{(A, I)\} \cup \{(A, \{\hat{\mathtt{cl}}\} \cup I)\} \\ \quad \textit{if } (A,I) \in \mathtt{MS}(\mathcal{P}) \textit{ for some rule index set } I. \\ \mathtt{MS}(\mathcal{P}) \cup \{(A, \{\hat{\mathtt{cl}}\})\} \\ \quad \textit{otherwise} \end{cases}$$

Now we use definiton 8 in algorithm 2.

$$\texttt{dercando}(X, Y, Z, \{\{\emptyset\}\}) \leftarrow memID(y), memStatus(\{y\}, X) \quad (17)$$

$$\texttt{dercando}(\{y, z, \{y\}\}, X, +, \{\emptyset\}) \leftarrow isAName(y), isAName(z), memMother(y, z), memStatus(\{y, z\}, X) \quad (18)$$

$$\texttt{dercando}(U, \{\{dlS, br, brTOC\} \mid X\}, +, \{\{\{\emptyset\}\}\}) \leftarrow memId(y), memStatus(\{y\}, X), y \in X \quad (19)$$

$$\texttt{do}(U, \{\{dlS, br, brTOC\} \mid X\}, +, \{\{\{\{\emptyset\}\}\}\}) \leftarrow memId(y), memStatus(\{y\}, X), y \in X \quad (20)$$

Algorithm 2 materialization algorithm

INPUT: An ABAC policy $\mathcal{P}$
OUTPUT: A materialization structure $\texttt{MS}(\mathcal{P})$ for $\mathcal{P}$
Base step: (materializing strata 0)
$\texttt{MS}_0 = \{(H(\vec{c}), \{\hat{\texttt{cl}}\})$: where $H(\vec{c})$ is a valid instance of the base predicate in the rule cl with index $\hat{\texttt{cl}}\}$.
Inductive Step: (materializing strata n+1)
$\texttt{MS}_{n+1} = \texttt{MS}_n \oplus \{(H(\vec{c}), \hat{\texttt{cl}})$ where $\texttt{cl} = H \leftarrow C, B$ is a rule cl with $\Phi_{n+1} \uparrow (\mathcal{P})(H(\vec{c}) = T$ satisfying $(b(\vec{c}, \vec{c'}), I) \in \texttt{MS}_n$ for some index set I for each $b \in B$ and $C(\vec{c}, \vec{c'})$ is valid$\}$. Here $\vec{c'}$ is the vector of extra constants that may be required to fully instantiate the body of cl.

Theorem 3 show that the materialization structure $\texttt{MS}(\mathcal{P})$ created using algorithm 2 for an ABAC policy $\mathcal{P}$ is correct according to definition 7.

THEOREM 3 (CORRECTNESS OF ALGORITHM 2). *Let $\mathcal{P}$ be an ABAC policy, and $\texttt{MS}_i(\mathcal{P})$ be its materialization structure created by algorithm 2 at stage i. Then, $\bigcup_{j \leq i} \texttt{MS}_j(\mathcal{P})$ correctly models $\Phi_I \uparrow (\mathcal{P})$.*

Proof: See the appendix.

Now we show the materialization structure for library policy given in example 2.

EXAMPLE 4 (MATERIALIZING POLICIES IN EXAMPLE 2). *As stated, the materialization structure created for the policy in example 2 is empty, because there are no base facts. Now suppose we enrich the policy with the three additional base facts* isAName(alice), isAName(bob), isMother(alice, bob), memStatus({alice, bob{alice}},{login}). *The first two recognize names and the last predicate says that credential set* {alice, bob,{alice}} *entitles the holder to the privileges set* {login} *in the digital library service. Consequently due to rule (4),* cando*({alice,bob,{alice}},{login}) is materialized by algorithm 2, using (say) the rule number 4 as the index. Following this indexing convention, we can use rule (5), to get that* cando*({alice,bob,{alice}}, {login},+,{∅}) is materialized with the rule set (4). Repeating this process, we get the following as a part of the materialization structure.*

5.2.1 Comparing Materialization Structures for two an three Valued Models

Materialization structures have been developed for logic programming based access control policies in the past, for example in [JSSS01]. However, such work materialized relations defined by logic programs - and not constraint logic programs. Consequently, corresponding materialization structures differ in two ways. The first is that three valued semantics were not considered in most logic programming systems, as there were no floundering queries. Therefore complications arising out of the *undefined* ($\perp$) truth value was not considered in the past. Consequently, definitions and theorems were stated and proved using classical satisfaction relations of fix-point theory. [JSSS01] is a case in point. Secondly, in our semantics, the ω closure of the three-valued consequence operator Φ does not constitute a fixed-point. This accounts for the remarkable difference in the details of proofs of corresponding facts. Consequently, our materialization structure construction does not compute fix-points for recursively defined predicates. But that leaves us with the disadvantage of a materialization algorithm that may take ω steps to complete. The next section shows that program transformation provides a manageable workaround for this problem.

5.3 Semantics, Program Transformations and Materializations

Algorithms 1 and 2 have the property that if $\Phi_n \uparrow (H(\vec{c})) = T$, then $H(\vec{c})$ becomes a rule in the transformed program, and $(H(\vec{c}), I) \in \texttt{MS}(\mathcal{P})$ for some rule index set I. That is, the three-valued immediate consequence operator, program transformation and the materialization produces exactly the same *valid* instances of reserved predicates. The basis for this observation is the fact that algorithms 1 and 2 are based on the step-wise construction of the three-valued Kripke-Kleene model of a Horn ABAC policy. Next we formally state and prove this fact.

THEOREM 4. *For any Horn ABAC policy $\mathcal{P}$, instantiated predicate $A(\vec{c})$ and integer n, $\Phi_n \uparrow (\mathcal{P})(A(\vec{c})) = T$ iff the n^{th} program transformation has $A(\vec{c})$ as a rule iff $(A(\vec{c}), I) \in \texttt{MS}_n$ for some rule index set I.*

Proof: See the appendix.

As stated, our ongoing research addresses extending theorem 4 to non-Horn clauses. Now we compute the three equivalent computations for the policy in example 4.

EXAMPLE 5. *The table given in example 4 show each predicate that is being materialized at each stage. Now we show the stages of computation for the bottom-up Kripke-Kleene mode, the policy transformations and the materialization.*

Stage 0: isAName(alice),isAName(bob), memMother(alice,bob), memStatus({alice,bob}, {login}) *are valid instance. Thus, they become instances of valid rules, thereby $\Phi_0(\mathcal{P})$ assigning truth value T to them. Because they are valid instances, algorithm 2 materializes them at stage 0. Coincidentally, algorithm 1 already has these predicate instance as valid rules.*

Stage 1: *Rule (4) can be instantiated to give* cando $(\{alice, bob\{alice\}\}, \{login\}, +, \{\emptyset\}) \leftarrow$

Stage (=n)	Materialized instances (members of $\mathtt{MS}_n$)
0	(isName(alice),{0.1}),(isName(bob),{0.2}),(isMother(alice,bob){0.3}), (memStatus({alice,bob{alice}},{login},{0.4}))
1	(cando({alice,bob,{alice}},{login},+,{∅}),{4})
2	(dercando({alice,bob,{alice}},{login},+,{{∅}}),{5})
3	(dercando({alice,bob,{alice}},{login,{dlS,br,brTOC}},+,{{{∅}}}),{7})
4	(do({alice,bob,{alice}},{login,{dlS,br,brTOC}},+,{{{{∅}}}}),{12})

$isAName(alice)$, $isAName(bob)$, $memMother(alice, bob)$, $memStatus(\{alice, bob\{alice\}\}, \{login\})$. Notice that the antecedents of this rule instance are all true under $\Phi_0(\mathcal{P})$. Therefore, the head of the rule cando($\{alice, bob\{alice\}\}, \{login\}, +, \{\emptyset\}$) *evaluates to be true under the three valued (Kleene) truth table. Thus,* $\Phi_0(\mathcal{P})$ (cando($\{alice, bob\{alice\}\}, \{login\}, +, \{\emptyset\}$)) $= T$.

As stated in example 4, algorithm 2 materializes cando ($\{alice, bob, \{alice\}\}, \{login\}, +, \{\emptyset\}$) *is materialized due to the reason that $\Phi_n(\mathcal{P})$ evaluates it to be true. Coincidentally, applying unfolding to* cando($\{alice, bob\{alice\}\}, \{login\}, +, \{\emptyset\}$)$\leftarrow isAName(alice)$, $isAName(bob)$, $memMother(alice, bob)$, $memStatus(\{alice, bob\{alice\}\}, \{login\})$ *with respect to $isAName(alice)$, $isAName(bob)$, $memMother(alice, bob)$ and $memStatus(\{alice, bob\{alice\}\}, \{login\})$ results in* cando($\{alice, bob\{alice\}\}, \{login\}, +, \{\emptyset\}$) *being evaluated to be true.*

As shown in example 5, the three-valued bottom-up model construction, materialization and unfolding all give the same facts. Thus, if we know the rank of the predicate instance we would like to evaluate say n, all we need to do is to unfold the policy n times. If the instance of interest (usually a reserved predicate such as do) is not there, then it must be false, as theorem 4 guarantees so. We can also make this procedure more efficient by only unfolding *relevant* rules. Our ongoing work in this aspect also addresses the issues of rule insertion, deleting and permission revocation [WJPPH03].

6. CONCLUSIONS

Requesting remote services in an identity-less open system requires that sets of attributes be presented in order to gain accesses to resources. In order to do so, we propose a stratified logic programming based framework to specify ABAC policies where collections of attribute and service options are modeled as sets in a computable hereditarily finite set theory. Our policies are flounder free, consistent and complete. In order to enhance runtime performance, we transform ABAC polices so that rewritten policies have the same runtimes as executing materialized rules. Our ongoing work explore other computable set theories and efficient implementations.

7. REFERENCES

E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, February 2003.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 134–143. ACM Press, 2000.

Piero Bonatti and Pierangela Samarati. A uniform framework for regulating service access and information release on the web. *Journal of Computer Security*, 10(3):241–272, 2002.

Steve Barker and Peter J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transactions on Information and System Security*, 2004. to appear.

David Chan. Constructive negation based on the completed databases. In R. A. Kowalski and K. A. Bowen, editors, *Proc. International Conference on Logic Programming (ICLP)*, pages 111–125. The MIT Press, 1988.

David Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proc. North-American Conference on Logic Programming*, pages 477–489. The MIT Press, 1989.

Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Transactions of Programming Languages and Systems*, 22(5):861–931, 2000.

Agostino Dovier, Alberto Policriti, and Gianfranco Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundamenta Informaticae*, 36(2/3):201–235, 1998.

Agostino Dovier, Carla Piazza, and Gianfranco Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. Technical Report Quaderno 235, Department of Mathematics, University of Parma, Italy, 2000.

Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Constructive negation and constraint logic programming with sets. *New Generation Comput*, 19(3):209–256, May 2001.

Sandro Etalle and Maurizio Gabbrielli. Transformations of clp modules. *Theoretical Computer Science*, 166:101–146, 1996.

Francois Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.

Melvin C. Fitting and Marion Ben-Jacob. Stratified, weak stratified, and three-valued semantics. *Fundamenta Informaticae, Special issue on LOGIC PROGRAMMING*, 13(1):19–33, March 1990.

Francois Fages and Roberta Gori. A hierarchy of semantics for normal constraint logic programs. In

Algebraic and Logic Programming, pages 77–91, 1996.

Melvin C. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

Melvin C. Fitting. Fixedpoint semantics for logic programming. *Theoretical Computer Science*, 278:25–31, 2002.

Joxann Jaffar and Jean-Louise Lassez. Constraint logic programming. *Proceedings of Principles of of Programming Languages*, pages 111–119, 1987.

Sushil Jajodia, Pierangela Samarati, Maria Louisa. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, June 2001.

Dexter C. Kozen. Set constraints and logic programming. *Information and Computation*, 142:2–25, 1998. Article No IC972694.

Kenneth J. Kunen. *Set theory: an introduction to independence proofs*. Elsevier North-Holland, 1980.

Kenneth J. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):298–308, December 1987.

N. Li, J.C. Mitchell, and W.H. Winsborough. Design of a role-based trust management framework. In *Proc. IEEE Symposium on Security and Privacy, Oakland*, pages 114–130, 2002.

Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proc. IEEE Symposium on Security and Privacy, Oakland*, May 2002.

Michael J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

Alberto Petterossi and Maurizio Proietti. *Transformation of Logic Programs*, volume 5, chapter Handbook of Logic in Artificial Intelligence and Logic Programming, pages 697–787. Oxford University Press, 1998.

R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Febraury 1996.

Peter J. Stuckey. Constructive negation for constraint logic programming. In *Logic in Computer Science*, pages 328–339, 1991.

Peter J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Logic Programming Conference*, pages 127–138, 1984.

Duminda Wijesekera, Sushil Jajodia, Francesco Parisi-Presicce, and Asa Hagstrom. Removing permissions in the flexible authorization framework. *ACM Transactions of Database Systems*, 28(3):209–229, September 2003.

T. Yu, M. Winslett, and K.E. Seamons. Prunes: an efficient and complete strategy for automated trust negotiation over the internet. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 210–219. ACM Press, 2000.

T. Yu, M. Winslett, and K.E. Seamons. Interoperable strategies in automated trust negotiation. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 146–155. ACM Press, 2001.

T. Yu, M. Winslett, and K.E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):1–42, 2003.

APPENDIX

A. PROOFS

Proof of Lemma 1: To prove the first claim, according to definition 1 the reserved predicates are `cando`, `dercando` and `do`. We consider each of them now.

`cando:` `cando`$(-,-,-,\{\emptyset\})\leftarrow B$ where B consists of non-reserved predicates or is empty. This is the only allowed form of `cando` in a rule head. Thus, R(`cando`$(-,-,-,\{\emptyset\})) = 1$ and R(b) $= 0$ for any predicate b in B.

`dercando:` According to the third rule in definition 1, if `dercando`$(-,-,-,Z)$ is in the body and `dercando`$(-,-,-,Z')$ is in the body, then $Z = \{Z'|U\}$ for some set U. Hence by definition 5, $R($`dercando`$(-,-,-,Z)) \geq 1 + R($`dercando`$(-,-,-,Z'))$.

`do:` The same argument applies for rules with a `dercando`$(-,-,+,Z)$ head. The only rule with a `dercando`$(-,-,+(-),Z)$ head is `do`$(X,Y,-,\{Z\})\leftarrow \neg$`do`$(X,Y,+,Z)$. Thus, $R($`do`$(X,Y,-,\{Z\})) = 1 + R($`do`$(X,Y,+,Z))$.

Now we use the first claim to justify the second. Suppose that $(A\cup\{p(\vec{s})\},C) \rightarrow_1 (A\cup B, C\cup C"\cup\{\vec{s}=\vec{t}\})$ is a one-step derivation using the ABAC rule $p(\vec{t})\leftarrow B, C"$ is a rule in $\mathcal{P}$ and $p(\vec{s})\leftarrow B, C"$ is a fresh instance of $p(\vec{s})\leftarrow B, C"$. To prove that $R(A\cup\{p(\vec{s})\}) > R(A\cup B)$, suppose $R(A\cup\{p(\vec{s})\}) = (k_m,\ldots,k_0)$, $R(P) = t < m$, and $R(B) = (t_u,\ldots,t_0)$ where $u < t$. Then $R(A\cup B) = (k_m,\ldots,k_{t+1},k_t-1,k_t,\ldots,k_i+t_i,\ldots,k_0)$. In the lexicographical ordering, $(k_m,\ldots,k_0) > (k_m,\ldots,k_{t+1},k_t-1,k_t,\ldots,k_i+t_i,\ldots,k_0)$, implying $R(A\cup\{p(\vec{s})\}) > R(A\cup B)$. ■

Proof of Theorem 1: Suppose $(A,C) \rightarrow_1 (A_0,C_0) \rightarrow_1 (A_1,C_1),\ldots$ is an infinite sequence of one-step reductions. Then by lemma 1, $R(A,C) > R(A_0,C_0) > \ldots$ is an infinite descending sequence, contradicting the well-foundedness of the rank function. This is a contradiction, as the lexicographical ordering on integers is well-founded. ■

Proof of Theorem 3: The proof follows from the fact that at each step i of the construction in algorithm 2 ensure that $\mathtt{MS}_i(\mathcal{P})$ correctly materializes $\Phi_i \uparrow (\mathcal{P})$. Consequently, our proof works by induction on i - the strata of the instantiation of the fourth varibale of reserved predicates.

Strata 0: By the definition of $\Phi_\omega \uparrow (\mathcal{P})$, $\Phi_0 \uparrow (\mathcal{P})(H(\vec{c})) = T$ iff $H(\vec{c}) \in \mathcal{P}$. By algorithm 2, $(H(\vec{c}),\{i\}) \in \mathtt{MS}_0(\mathcal{P})$ where $\{i\}$ is the index of $H(\vec{c})$ chosen according to some rule indexing schema. Thus $\Phi_\omega \uparrow (\mathcal{P})(H(\vec{c})) = T$ iff $H(\vec{c}) \in \mathtt{MS}_0(\mathcal{P})$, satisfying the first condition for $\Phi_\omega \uparrow (\mathcal{P})$ to correctly model

$\mathcal{P}$. The second condition is vacuously satisfied as, by definition 1 the only rules in which base predicates appear as heads are the ones with empty bodies.

Strata n+1: Suppose the inductive hypothesis holds for all instances of atoms of lower ranks and $H(\vec{c})$ is an instance of a reserved predicate of rank $n+1$. Then there must be at least one rule in which the body has at least one reserved predicate with rank n. Choose any such rule `cl`, say $H \leftarrow C, B$. There are two cases to consider.
Case 1: $\Phi_n \uparrow (\mathcal{P})(B(\vec{c}, \vec{c'})) = T$. For the first condition, suppose $C(\vec{c}, \vec{c'})$ is satisfiable. Furthermore, $\Phi_n \uparrow (\mathcal{P})(b(\vec{c}, \vec{c'})) = T$. Thus by the inductive hypothesis $b(\vec{c}, \vec{c'}) \in \mathtt{MS}_n(\mathcal{P})$. Then by algorithm 2 $(H(\vec{c}), I) \in \mathtt{MS}_n(\mathcal{P})$ for a set I of rule indices. Thus the first condition for $\mathtt{MS}(\mathcal{P})$ correctly materializing $\mathcal{P}$ is met by satisfying both sides of the by-implication stated in definition 7.

For the second condition, consider any rule `cl` where $H(\vec{c})$ has rank $n+1$ and $\Phi_\omega \uparrow (\mathcal{P})(B(\vec{c}, \vec{c'})) = T$. Then it could be shown by induction that $\Phi_n \uparrow (\mathcal{P})(B(\vec{c}, \vec{c'})) = T$. Now by repeating the previous part of the argument, if $C(\vec{c}, \vec{c'})$ is satisfiable, then $(H(\vec{c}), I) \in \mathtt{MS}(\mathcal{P})$ where $\hat{\mathtt{cl}} \in I$, for some rule set index I that contain $\hat{\mathtt{cl}}$.

Case 2: $\Phi_n \uparrow (\mathcal{P})(B(\vec{c})) \neq T$. This could be so due to one of two factors: (1) $\Phi_n \uparrow (\mathcal{P})(b(\vec{c}, \vec{c'})) \neq T$ for some atom $b \in B$. (2) $C(\vec{c}, \vec{c})$ does not hold. If (1) is the case, then by the inductive hypothesis, $(b, I) \notin \mathtt{MS}_n(\mathcal{P})$ for any index set I. In either case, the inductive step of algorithm 2 does not add $H(\vec{c}, \{\hat{cl}\})$ into $\mathtt{MS}_{n+1}(\mathcal{P})$. Therefore the first condition of correctness criteria in definition 7 holds.

The second condition is satisfied because, as stated the inductive step of algorithm 2 does not add any steps when $\Phi_n \uparrow (\mathcal{P})(B(\vec{c})) \neq T$. ■

Proof of Theorem 4: We prove by induction on the rank of the predicate instance $A(\vec{c})$.

The Base Case $R(A(\vec{c})) = 0$: In this case, the predicate is a base predicate. Thus either $A(\vec{x})$ or $A(\vec{c})$ appears in $\mathcal{P}$, or A only appear in bodies of rules with reserved word heads. If the first case occurs, $\Phi_\omega(\mathcal{P})(A(\vec{c'})) = T$ for all constant vectors $\vec{c'}$. If the second case occours then $\Phi_\omega(\mathcal{P})(A(\vec{c})) = T$ for only those combinations. As algorithms (1) and (2) show these are the only conditions under which an instantiated zero ranked reserved predicate becomes a rule and materialized respectively.

The Inductive Case $R(A(\vec{c}) = n+1$: Suppose the claim is true for all instantiated predicates with ranks $m \leq n$, and $R(A) = n+1$. Under stated assumptions, $A(\vec{x})$ appears as a head in some rule $\mathtt{cl} = H \leftarrow C, B$, because otherwise it cannot have a positive rank, as $R(A(\vec{c})) = n+1 > 0$.

Now suppose $A(\vec{x})$ appears as a head in some rule $\mathtt{cl} = H \leftarrow C, B$ with the usual convention that C and B are the constraint and non-constraint predicates. Suppose $\Phi_{n+1}(A(\vec{c})) = T$. Then $C(\vec{c})$ must be valid and $\Phi_n \uparrow (\mathcal{P})(B(\vec{c})) = T$. Thus by algorithm 2, $(A(\vec{c}), I) \in \mathtt{MS}_{n+1}$ for some rule index set I satisfying $\mathtt{cl} \in I$. Furthermore, by the inductive hypothesis, $B(\vec{c}, \vec{c'})$ are rules in the n^{th} transformed policy. Therefore, by applying unfolding with `cl` we get that $A(\vec{c})$ is a rule at the $n+1^{th}$ transformed policy. The implication from the materialization to the truth in $\Phi_{n+1} \uparrow (\mathcal{P})$ follows trivially, as $A(\vec{c})$ gets materialized in the stage immediately after $b(\vec{c}, \vec{c'})$ gets materialized for all non-constraint predicates b in the body B. Now suppose $A(\vec{c})$ becomes a rule at the $n+1^{th}$ stage of algorithm 1. Then, $A(\vec{c})$ must be a result of applying unfolding to some rule, $A \leftarrow C, B$ say `cl`. Then, it must be that the ranks of each non-constraint predicate b of B must be less than or equal to n. Because if all their ranks were less than n, then $A(\vec{c})$ gets unfolded at most stage n. Then there must be a non-constraint predicate (say) b with $R(b(\vec{c}, \vec{c'})) = n$ Then by the inductive hypothesis $\Phi_n \uparrow (\mathcal{P})(B(\vec{c}, \vec{c'})) = T$, implying $\Phi_{n+1} \uparrow (\mathcal{P})(A(\vec{c})) = T$. ■