Flexible Access Control Policy Specification with Constraint Logic Programming

STEVE BARKER
King's College, University of London
and
PETER J. STUCKEY
University of Melbourne

We show how a range of role-based access control (RBAC) models may be usefully represented as constraint logic programs, executable logical specifications. The RBAC models that we define extend the "standard" RBAC models that are described by Sandhu et al., and enable security administrators to define a range of access policies that may include features, like denials of access and temporal authorizations, that are often useful in practice, but which are not widely supported in existing access control models. Representing access policies as constraint logic programs makes it possible to support certain policy options, constraint checks and administrator queries that cannot be represented by using related methods (like logic programs). Representing an access control policy as a constraint logic program also enables access requests and constraint checks to be efficiently evaluated.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*security, integrity, and protection*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authorization*; *access control*; D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages.*

General Terms: Security.

Additional Key Words and Phrases: Role-based access control, constraint logic programming.

1. INTRODUCTION

Ensuring the security of the data that are contained in information systems has long been recognized as important. Users of a system must be authenticated to be legitimate users, and must only be permitted to retrieve and modify data in the ways that are authorized by an access control policy specification.

1.1 Motivations

The models and languages that have hitherto been proposed and used for representing access control policies have been limited in expressive power, and only

Author's addresses: S. Barker, Department of Computer Science, King's College, University of London, Strand, London, U.K, WC2R 2LS, `steve@dcs.kcl.ac.uk`. P.J. Stuckey, Dept. of CS & SE, University of Melbourne, 3010, Australia, `pjs@cs.mu.oz.au`.

allow simple types of access policy to be defined for protecting simple types of information systems. There is, however, a pressing need for more expressive access control models and languages to enable organizations to flexibly represent access policies for protecting the increasingly diverse and complex types of systems in use today. Moreover, these models and languages must be formally well-defined so that access control policies can be proven to satisfy certain properties (e.g., that no unauthorized access to data is permitted and that no authorized access is denied).

Despite the importance of access control, only limited access control facilities are supported in existing systems. In the case of relational DBMSs, for example, the standard approach has been to support *discretionary access control (DAC)* policies [Date 2000]. However, DAC policies are often inappropriate for organizations to use [Ferraiolo et al. 1995]. Specific DAC-based models, like SQL's GRANT-REVOKE model [Date 2000], have their distinct shortcomings [Barker and Rosenthal 2001]. Partly in response to the limitations of DAC, there has been considerable interest in *role-based access control (RBAC)* policies [Sandhu et al. 1996]. However, although RBAC facilities are beginning to be supported in commercial systems (e.g., ORACLE version 8), the facilities that are offered by existing systems do not enable SAs to represent a wide range of access control policies.

1.2 Candidate Approaches

To overcome the limitations of existing languages for formulating access control requirements, applications programs written in a Turing complete language may be used. However, embedding (all or part of) an access control policy in an application program that is written in low-level procedural code is unsatisfactory as this makes it difficult for a *security administrator (SA)* to reason about the effects of the policy, the modification and maintenance of the policy is complicated, and proving properties of the policy is problematic. Applications written in a high-level declarative language, like pure PROLOG [Sterling and Shapiro 1994], have the attractions of being easily understandable and easy to modify and reason about. However, efficiency concerns arise. To help to ensure adequate performance, operational features are often used in applications (e.g., the *cut* in (impure) PROLOG), but such features compromise the very attractions of using declarative languages (e.g., their separating out of logic and control).

An alternative to using a Turing complete language for representing access control requirements is to use special purpose languages that provide a SA with features for formulating non-standard access control policies (e.g., the temporal authorization language described in [Bertino et al. 1998]). However, special purpose languages only provide a SA with small sets of pre-defined operators. In practice, several special purpose languages are likely to be required to enable a SA to represent the range of access control policies that need to be specified for real organizations.

1.3 Our Approach and Contributions

In the sequel, we consider the use of *constraint logic programming (CLP)* for formulating and implementing RBAC and *temporal RBAC (TRBAC)* policies.[1]

[1]Although we consider RBAC and TRBAC policies in this paper, many different types of access policies may be represented in the way that we describe (e.g., constraints may be used to represent

Although we have previously suggested using logic programs to represent access control requirements [Barker 2000a], we believe that there are a number of reasons to consider using CLP, rather than logic programming, for the specification and implementation of access control policies. By using CLP, it is possible to exploit the attractions of using a declarative approach for representing access control policies (e.g., conciseness of specification, easily understandable and maintainable specifications of requirements, strong technical results that enable properties of a policy to be proved, ...); constraint logic programs enable certain policies to be formulated that cannot be satisfactorily represented by using logic programs (see below); and, for evaluating access requests with respect to a specification of the access control policy, we get competitive performance with special purpose algorithms by exploiting the efficiency of logic programming in database querying and efficient constraint-solving methods (see Section 10).[2] The facts that CLP languages can deal with uninstantiated arithmetic variables in computations (logic programming language are limited in this respect), and can return sets of constraints as answers can be usefully exploited in the management of an access control policy, for enabling a SA to query an access policy, and for constraint checking on the policy (see below).

1.4 Preliminary Notions

In the context of access control, in order for a user to be able to perform an action on an object a user is usually required to have an *access privilege* that permits the user to perform the requested action on the object. In the ensuing discussion, we will use U, O and A to, respectively, denote an arbitrary user, object and access privilege, and we assume that an individual has been authenticated to be a legitimate user of a system at the time at which the individual makes a request to access a data object that is contained in the system.

In subsequent sections, we demonstrate our approach by showing how a range of *open*, *closed* and *hybrid* RBAC and TRBAC policies may be represented. These classes of access policies differ in terms of the authorizations they admit. An *authorization* is usually expressed as a triple $\langle U,A,O \rangle$. A *positive authorization* [Castano et al. 1995] is used to specify that a user U has A access on a data object O; a *negative authorization* [Castano et al. 1995] is used to specify that U is prohibited the A access on O. Positive authorizations enable a SA to enforce a *closed access policy* [Castano et al. 1995]; negative authorizations enable a SA to enforce an *open access policy* [Castano et al. 1995]. Policies that have open and closed features (i.e., policies that include positive and negative authorizations) are called *hybrid* policies. Temporal forms of authorization are used to specify an interval of time for which an authorization is to hold.

If positive and negative authorizations may be expressed in a specification of an access policy then it is possible for a user to be recorded as having A access on an object O and as being denied A access on O. A constraint may be imposed on a

---

the user security clearances and object security classifications used in *mandatory access control (MAC)* models [Castano et al. 1995]).

[2]The constraints that we use to specify access control policies involve equations, inequations and disequations involving integers; it is well known that efficient (albeit incomplete) techniques exist for handling these types of constraints [Marriott and Stuckey 1998].

policy specification to prevent this type of conflicting pair of authorizations being defined (see below). Alternatively, a *conflict resolution strategy* [Castano et al. 1995] may be used to resolve the inconsistent authorizations by specifying which of the authorizations in a conflicting pair ought to hold.

1.5 Related Works

In work related to ours, Woo and Lam [Woo and Lam 1993] propose the use of *default logic* [Reiter 1980] for representing access control policies. Although the expressive power of default logic enables a wide range of access policies to be formulated, Woo and Lam's approach involves computing *extensions* [Woo and Lam 1993] of arbitrary default theories. The computation involved is $\mathcal{NP}$-complete in the case of propositional default theories and is worse than undecidable in the first-order case. Hence, although the language that Woo and Lam suggest is attractive in terms of allowing a SA to formulate a wide range of access control requirements, the computational problems that can arise, when access requests are evaluated with respect to policies expressed in this language, makes the approach impractical to use (in its full generality).

To avoid the problems of non-decidability and non-implementability that arise in Woo and Lam's approach, a practical subset of *clause form logic* [Kowalski 1979], *locally stratified logic* [Przymusinski 1988], is used by Jajodia et al. [Jajodia et al. 2001] to specify a range of high-level access control policies for protecting computer-based "systems". However, there is no treatment of temporal authorizations in Jajodia et al's proposals, there is no discussion of how SAs might extract information from a policy specification, and the proposal is based on a rather ad hoc access control model (that exhibits both discretionary and role-based features). Moreover, flexible policy specification is interpreted by Jajodia et al. in terms of the flexible specification of open, closed and hybrid policies (with various conflict resolution strategies in the latter case). In our approach, open, closed and hybrid policies (with the same conflict resolution strategies that Jajodia et al. describe) may be formulated, as well as policies that enable temporal authorizations and other forms of constraints on access to be flexibly specified (e.g., our proposal permits a variety of permission/denial inheritance policies to be formulated).

In more recent work on the representation of multiple access policies, Bertino et al. [Bertino et al. 2002] build on the logic programming approaches described in [Jajodia et al. 2001] and [Barker 2000b] by showing how C-Datalog [Greco et al. 1992] may be profitably used to express various access control policies. In common with the proposal described in [Bertino et al. 2002], the work that we describe is concerned with the exploitation of declarative and expressive languages for enabling multiple access models to be formulated. However, unlike [Bertino et al. 2002], we use constraint logic programming for multi-policy specification and implementation. As we will show, using constraints to represent access control policies has certain attractions, and we demonstrate how a range of temporal access control models may be flexibly specified by using CLP.

In [Bertino et al. 2000] and [Barker 2000a], RBAC models are presented that enable temporal authorizations to be expressed in logic languages. The model in [Bertino et al. 2000] is based on the notion of *sessions* [Sandhu et al. 1996], and uses a trigger-based language to formulate access control requirements. Unfortunately, the model described in [Bertino et al. 2000] has certain limitations (e.g., limitations in terms of the constraints that may be imposed on TRBAC policies), and the language that is proposed is not well-suited for proving properties of a specification of access control requirements. In [Barker 2000a], a rich family of temporal RBAC (TRBAC) models is described and is represented by using logic programs. Unfortunately, these models are expensive to compute with [Barker 2001], and certain forms of query evaluation on these programs are problematic because their evaluation may involve comparison operators with uninstantiated variables. The TRBAC models that we describe in this paper are more powerful than those in [Barker 2000a], they can be represented in a more succinct way, and constraint representation and checking are simpler.

The temporal access control model described in [Bertino et al. 1998] is a very important contribution in the development of more expressive forms of access control model. However, this model does exhibit certain shortcomings. For instance, the Bertino et al. model is based on a rather ad hoc mixture of DAC and RBAC features, it includes a language that is not effectively implementable in its full generality, restrictions are imposed on the type of access policies that can be expressed (e.g., a closed policy is assumed and a fixed "denials take precedence" conflict resolution policy is adopted), and specifications of an access policy using the language that Bertino et al. propose are not guaranteed to define a unique set of authorizations. To ensure that a unique set of authorizations is defined, complexity is introduced into the model in the form of a *critical sets* detection algorithm. There is no discussion in [Bertino et al. 1998] of the importance of maintaining a history of authorizations, and no discussion of the constraints that may be imposed on access policy specifications.

In contrast to the model described by Bertino et al., our approach is effectively implementable, specifications of access policies can be guaranteed to define a unique set of authorizations, no unnecessary restrictions are imposed on the types of policy that can be formulated, and authorization histories can be easily accommodated. Moreover, unlike existing proposals for implementing multiple access policies (e.g., [Bertino et al. 1998] and [Jajodia et al. 2001]), the operational semantics that we describe in the sequel enables access requests to be efficiently evaluated, with respect to a policy specification, without requiring that any special means be implemented to enable authorizations to be materialized.[3] We also choose to formulate policies that are based on a reasonably well-defined set of "standard" RBAC models, the RBAC models described in [Sandhu et al. 2000]. Henceforth, we refer to these models as the *SFK* models. Another contribution of our work is to formally specify the (previously informally specified) SFK models. We also propose a number of extended forms of the SFK models.

[3]Materializing authorizations is appropriate only under certain assumptions e.g., the assumption that a policy includes few "parametric" *derivation rules* [Bertino et al. 1998], and the assumption that sets of authorizations are reasonably static. Because sessions play an important part in our access policies, sets of authorizations will change in a highly dynamic manner.

1.6 Organization

The rest of the paper is organized thus. In Section 2, we describe some notions in CLP that are relevant to our approach. In Section 3, we formally define some key concepts relating to RBAC and the *RBAC programs* (constraint logic programs) that we use to represent access policies. In Section 4, we consider the representation of the level 2A RBAC model from [Sandhu et al. 2000] as an RBAC program. In Section 5, we define some extended forms of the "standard" RBAC models from [Sandhu et al. 2000], and we show how policies that are based on these models may be represented by using RBAC programs. In Section 6, we describe temporal RBAC programs: *TRBAC programs*. In Section 7, we consider the evaluation of access requests with respect to TRBAC programs. In Section 8, we consider constraint checking on TRBAC programs. In Section 9, we show how SAs may extract information about a TRBAC program by querying the program. In Section 10, we discuss some performance issues relating to implementations of our TRBAC programs. Finally, in Section 11, some conclusions are drawn.

2. CONSTRAINT LOGIC PROGRAMMING: BASIC CONCEPTS

In this section we briefly describe some essential concepts in constraint logic programming that are relevant in our approach. For further details on constraint logic programs we refer the reader to [Marriott and Stuckey 1998].

We begin by defining the key notions of a *primitive constraint* and a *constraint*.

DEFINITION 2.1. *A primitive constraint c has the form $p(t_1, \ldots, t_n)$ where p is a (predefined) constraint relation of arity n and $t_1, \ldots, t_n$ are terms.*

DEFINITION 2.2. *A constraint C is a conjunction of primitive constraints $c_1 \wedge \cdots \wedge c_k$ where $\wedge$ is the logical 'and' operation.*

In this paper we will be interested in two kinds of constraints. The first kind are equational constraints over sets of constants that represent roles, users, access privileges and objects. The only terms of interest are constants and variables, and the only primitive constraints of interest are of the form $t_1 = t_2$ and $t_1 \neq t_2$. Throughout the paper, constants are denoted by symbols that appear in the lower case; variables are denoted by symbols that start with an upper case letter. Example primitive constraints are $User = fred$ and $Role \neq doctor$.

The second kind of constraints of interest are constraints over non-negative integers. Terms are constructed from variables, integer constants and the arithmetic operators $\times, \div, +, -$ and *mod*; primitive constraints are restricted to the predefined constraint relations $=, \geq, \leq, >, <$ and $\neq$ (which are all binary). Example primitive constraints are $T = 3$, $T \geq U$ and $Y = (T \div 10000)\ mod\ 10000$.

A *positive literal* is either a primitive constraint or an atom. An *atom* has the form $p(t_1, ..., t_n)$ where p is a user-defined predicate symbol and $t_1 \ldots t_n$ are terms. A *negative literal* is a negated atom written *not* A (where A is an atom). A *literal* is an atom or its negation. A *goal* G is a sequence of literals, $L_1, \ldots, L_m$. A *rule* is of the form $H \leftarrow L_1, \ldots, L_m$ where H, the *head* of the rule, is an atom and $L_1, \ldots, L_m$ is the *body*. If $m = 0$ then we write the rule simply as H and refer to the rule as a *fact*. We let $vars(O)$ return the set of variables appearing in an object O, which might be a literal, goal or rule.

DEFINITION 2.3. *A (constraint logic) program is a finite set of rules.*

DEFINITION 2.4. *A definite program is a program that does not include any negative literals.*

We can understand the meaning of a definite program, as a conjunction of logical formulas. Each rule, $H \leftarrow L_1, \ldots, L_n$, corresponds to the first-order statement (clause) $\tilde{\forall}(H \leftarrow L_1 \wedge \cdots \wedge L_n)$ where $\tilde{\forall}$ is universal closure. That is, the head H is true (proved) if the conjunction of the literals appearing in the body $(L_1, \ldots, L_n)$ is true (proved).

In order to represent access policies with denials, we will need to go beyond definite programs to allow programs with negation. That is, programs with rules of the form $H \leftarrow L_1, \ldots, L_n$ where L_i $(1 \leq i \leq n)$ is a positive or a negative literal. However, we restrict our attention to *stratified programs*. The key definitions follow next.

DEFINITION 2.5. *In the rule $H \leftarrow L_1, \ldots, L_n$, H is said to positively (resp. negatively)* refer to *the literal $L_i, 1 \leq i \leq n$, if L_i is positive (resp., negative) in the body of the rule.*

DEFINITION 2.6. *A program dependency graph has an edge labelled $+$ (resp., $-$) from H to L_i if H refers to L_i positively (resp., negatively) in a rule.*

DEFINITION 2.7. *A normal program is stratified if and only if its program dependency graph contains no cycle with a negative edge.*

DEFINITION 2.8. *A normal program is recursion-free iff its program dependency graph is acyclic.*

An important property of the programs that we use for representing RBAC and TRBAC policies using CLP will be that they terminate. The simplest way to ensure termination is to restrict attention to recursion-free programs. Even with this restriction, we can represent complex access policy requirements. However, it is also possible to admit recursive programs and to use *termination analysis* [Genaim and Codish 2001; Speirs et al. 1997] to ensure that the program terminates for all queries of interest.

For a program Π with negation, we must map Π to slightly more complicated first-order formulas.

DEFINITION 2.9. *The definition of an n-place predicate symbol p in the logic program* Π *is the formula*

$$\forall X_1 \ \ldots \forall X_n \ p(X_1, \ldots, X_n) \leftrightarrow B_1 \vee \ldots \vee B_m$$

where $\vee$ is logical or, $\leftrightarrow$ is bi-implication and each B_i corresponds to a rule in Π *of the form*

$$p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_k$$

and B_i is

$$\exists Y_1 \ \ldots \exists Y_j \ (X_1 = t_1 \wedge \ldots \wedge X_n = t_n \wedge L_1 \ldots \wedge L_k)$$

where $Y_1, \ldots, Y_j$ are the variables in the original rule, and $X_1, \ldots, X_n$ are variables that do not appear in any rule. If there is no rule with head p, then the definition *of* p *is simply*

$$\forall X_1 \ \ldots \forall X_n \ p(X_1, \ldots, X_n) \leftrightarrow false$$

as $\bigvee \emptyset$ *is naturally considered to be* $false$*.*

Definition 2.10. *The (Clark) completion,* $\Pi^\star$*, of a logic program* Π *is the conjunction of the definitions of the user-defined predicates in* Π.

The operational semantics of a constraint logic program is given in terms of the "derivations" from goals. The execution of the goal determines under what conditions the goal holds (is proven). The execution relies on a *constraint solver*, *solv*, that determines which constraints may be satisfiable. The constraint solver *solv* is a function, $solv : C \mapsto BOOL$ (where C is a constraint and $BOOL = \{true, false\}$), that is defined in the following way:

$$solv(C) = \begin{cases} false, & \text{if } C \text{ is an unsatisfiable constraint,} \\ true, & \text{otherwise.} \end{cases}$$

We begin our description of the operational semantics of constraint logic programs by restricting our attention to definite programs. We then generalize the discussion to allow rules with negative literals. A key notion is that of a *state*.

Definition 2.11. *A* state *is a tuple* $\langle G \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} C \rangle$ *that contains the current literal sequence or goal* G *and the current constraint* C*.*

Definition 2.12. *A reduction is a transformation of a state* S_i *to a state* S_j *by the application of the rules in a program.*

At each reduction step, a literal in the goal is *selected* according to some fixed *selection rule* [Lloyd 1987], which is often left-to-right. If the selected literal L is a primitive constraint, and L is consistent with the current constraint C_{CURR}, then L is added to C_{CURR}. If L is inconsistent with C_{CURR} then the derivation "fails". If L is an atom then it is reduced by using one of the rules in its definition.

A state $\langle L_1, ..., L_m \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} C \rangle$ can be *reduced* with respect to a program Π as follows. Select a literal L_i ($i \in \{1, .., m\}$) then:

(1) If L_i is a primitive constraint and $solv(C \wedge L) \neq false$ then the state is reduced to $\langle L_1, ..., L_{i-1}, L_{i+1}, ..., L_m \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} C \wedge L_i \rangle$.
(2) If L_i is a primitive constraint and $solv(C \wedge L) = false$ then the state is reduced to $\langle \Box \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} false \rangle$, where $\Box$ represents the empty sequence.
(3) If L_i is an atom $p(s_1, ..., s_n)$ then the state is reduced to

$$\langle L_1, ..., L_{i-1}, s_1 = t_1, ..., s_n = t_n, B, L_{i+1}, ..., L_m \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} C \rangle$$

using a (renamed apart) version of a rule in Π of the form $p(t_1, ..., t_n) \leftarrow B$.
(4) If L is an atom $p(s_1, ..., s_n)$ in Π and there are no rules for p in Π then the state is reduced to $\langle \Box \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} false \rangle$.

Definition 2.13. *A derivation from a goal* G *in a program* Π *is a sequence of states* $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$ *where* S_0 *is* $\langle G \mathbin{\rule[-0.1ex]{0.15em}{1.5ex}} true \rangle$ *and there is a reduction from each* S_{i-1} *to* S_i*, using the rules in* Π*. The state* S_n *is called the final state.*

Definition 2.14. *A derivation* $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$ *from a goal* G *is finished if the goal in the final state* S_n *cannot be reduced.*

The final state in a finished derivation from G must have the form $\langle \Box \mid C\rangle$.

Definition 2.15. *If* C *is* $false$ *in the final state of a finished derivation then the derivation is said to be failed. Otherwise the derivation is successful.*

Definition 2.16. *If all derivations of a goal* G *for program* Π *are failed then* G *is finitely failed.*

Definition 2.17. *If* $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$ *is a successful derivation from a goal* G *and* C *is the constraint in the final state* S_n *then the answer for* G *is defined as* $\exists v_1 \cdots \exists v_n\ C$ *where* $\{v_1, \ldots, v_n\} = vars(C) - vars(G)$.

Definition 2.18. *The answers of a goal* G *for program* Π *are the answers arising from each successful derivation from* G.

We next extend the operational semantics to negative literals. The *constructive negation*[4] treatment of negative literals is defined as follows.

(5) If L_i is a negative literal of the form *not* $p(s_1, \ldots, s_n)$, let $C_1, \ldots, C_k$ be the answers to the state $\langle p(s_1, \ldots, s_n) \mid C\rangle$ and let $D_1 \vee \cdots \vee D_m$ be equivalent to $\neg(C_1 \vee \cdots \vee C_k)$ where each $D_j, 1 \leq j \leq m$ is a constraint and $\neg$ is classical negation. Then the state $\langle L_1, ..., L_m \mid C\rangle$ is reduced to $\langle L_1, ..., L_{i-1}, L_{i+1}, ..., L_m \mid C \wedge D_j\rangle$ for some j if $solv(C \wedge D_j) \neq false$. If $solv(C \wedge D_j) = false$ for all j then the state $\langle L_1, ..., L_m \mid C\rangle$ is reduced to $\langle \Box \mid false\rangle$.

The use of constructive negation requires that the negation of an existentially quantified constraint leads to a disjunction of constraints (*admissible closure* [Stuckey 1995]). Although the non-negative integer constraints that we use in our access control programs do not have the admissible closure property in general, an expressive subclass of these constraints can be naturally handled.

The key theoretical result that expresses the correspondence between the operational semantics and the three-valued logical consequences ($\models_3$) of the completion of the program is as follows.[5]

Proposition 2.19. *Let* G *be a goal and* Π *be a program. Then,* $\Pi^\star \models_3 G \leftrightarrow C$ *iff there are answers* $C_1, \ldots, C_k$ *to* G *with respect to* Π *(using constructive negation), such that* $C \leftrightarrow C_1 \vee \cdots \vee C_k$ *[Stuckey 1995].*

The complexity of three-valued consequences is not required if we restrict ourselves to programs where all goals terminate. For such programs we are guaranteed that the three-valued consequences and two-valued consequences coincide.

Proposition 2.20. *If* Π *is a program where all goals terminate then* $\Pi^\star$ *has a unique two-valued model, which is also its unique three-valued model.*

[4]This form of constructive negation assumes termination of every goal. For more general forms see [Stuckey 1995].

[5]Technically, this result also requires the use of a *fair* computation rule. See [Stuckey 1995] for details.

PROPOSITION 2.21. *The unique model of* $\Pi^\star$ *is also the perfect model [Przymusinski 1988] and unique stable model of* $\Pi^\star$ *[Gelfond and Lifschitz 1988].*

Although constructive negation is expensive to compute in general, there is a more efficient version of negation that is applicable if the constraints C fix all of the variables that appear in a negative literal. The *negation-as-failure* treatment of negative literals is defined as follows.

(6) If L_i is a negative literal of the form *not* $p(s_1, \ldots, s_n)$, and all solutions of C give the same value for each variable in $vars(L_i)$ then either (a) the state $\langle p(s_1, \ldots, s_n) \mid C\rangle$ has a successful derivation in which case $\langle L_1, \ldots, L_m \mid C\rangle$ is reduced to $\langle \Box \mid false\rangle$, or (b) the state $\langle p(s_1, \ldots, s_n) \mid C\rangle$ is finitely failed and $\langle L_1, \ldots, L_m \mid C\rangle$ is reduced to $\langle L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m \mid C\rangle$.

In computing with RBAC and TRBAC programs, we will always be able to use the efficient negation-as-failure approach for access control because the form of programs that we produce will ensure that negative literals are only selected when all of their variables are fixed. The following definitions and results are important in this respect and will be used in the sequel.

DEFINITION 2.22. *A rule* $H \leftarrow B_C, B_A, B_N$ *(where* B_C *are the primitive constraint literals,* B_A *are the atoms and* B_N *are the negative literals) is range restricted if for any constraint* C *fixing variables occurring in* B_A *we have that* $C \wedge B_C$ *fixes all of the variables occurring in the rule.*

DEFINITION 2.23. *A program is range restricted if all of its rules are range restricted.*

PROPOSITION 2.24. *Given a goal* G *where all variables are fixed and a range restricted program* Π*, there is a selection strategy where each negative literal has all variables fixed when selected [Lloyd 1987].*

DEFINITION 2.25. *A program* Π *is negation-as-failure safe for a set of goals* $\mathbf{G}$ *if in any execution of a goal* $G \in \mathbf{G}$ *it is the case that whenever a negative literal is selected then all of its variables are fixed by the current constraint.*

PROPOSITION 2.26. *A range restricted program is negation-as-failure safe for all variable-free queries [Lloyd 1987].*

Although some of the programs that we will use for representing TRBAC policies are not range-restricted they can be proven to be negation-as-failure safe for ground goals by using groundness analysis methods for (constraint) logic programs (see, for example, [Marriott et al. 1990]).

The result that follows supports the coincidence of the logical and operational semantics for negation-as-failure safe programs.

PROPOSITION 2.27. *Let* G *be a goal and* Π *be a program that is negation-as-failure safe for* G*. Then,* $\Pi^\star \models_3 G \leftrightarrow C$ *iff there are answers* $C_1, \ldots, C_k$ *to* G *with respect to* Π *(using negation as failure), such that* $C \leftrightarrow C_1 \vee \cdots \vee C_k$ *[Cavedon and Lloyd 1989].*

In later sections, we discuss the importance of the coincidence of the logical and operational semantics with respect to the type of programs that we use to represent access control policies.

Level	*Model*	*Functional Capabilities*
1	$RBAC_F$	*Users acquire permissions through roles* *Many-to-many user-role assignment* *Many-to-many permission-role assignment* *User-role review* *Simultaneous use of permissions on multiple roles*
2	$RBAC_{H2A}$	$RBAC_F$ + *Unrestricted role hierarchies*
	$RBAC_{H2B}$	$RBAC_F$ + *Restricted role hierarchies*
3	$RBAC_{C3A}$	$RBAC_{H2A}$ + *Separation of duties*
	$RBAC_{C3B}$	$RBAC_{H2B}$ + *Separation of duties*
4	$RBAC_{S4A}$	$RBAC_{C3A}$ + *Permission-role review*
	$RBAC_{S4B}$	$RBAC_{C3B}$ + *Permission-role review*

Table I. The SFK Models.

3. RBAC PROGRAMS: FEATURES AND FORMALIZATION

In this section, we begin by providing an overview of the key notions relating to our proposed extended forms of the SFK models from [Sandhu et al. 2000]. We call these models the *extended SFK models* (the *ESFK* models).

3.1 RBAC: Basic Concepts

The minimum requirements of any RBAC model are that it provides means for specifying that (i) access privileges on data objects are assigned to a role and that (ii) users are assigned to a role. The former type of assignment is referred to as *permission-role assignment*; the latter is referred to a *user-role assignment*. In addition to permission-role and user-role assignments, role hierarchies and constraints are important in RBAC [Sandhu et al. 2000].

In [Sandhu et al. 2000], Sandhu, Ferraiolo and Kuhn informally describe a set of "standard" levels of RBAC. The key features of the SFK models are summarized in Table I. We refer the reader to [Sandhu et al. 2000] for details of these models.

In the sequel, we will describe our ESFK models and the representation of access policies that are based on these models by using RBAC (constraint) programs. In particular, we will show how our RBAC programs enable open and hybrid access policies to be represented (with or without temporal constraints on access).[6] In our representation of RBAC programs, authorizations are only ever positive. However, this does not restrict the types of policy that can be represented. In our proposal, closed access policies are formulated by using permission-role assignments; open access policies are formulated by using *denial-role assignments*(defined below); and hybrid policies are formulated by using permission-role and denial-role assignments. Denial-role assignments are not admitted in the SFK models.

[6]The SFK models only enable non-temporal closed access policies to be represented.

3.2 RBAC Programs

For the specification of RBAC programs that we propose, a number of definitions are required. We note first that an RBAC program is defined on a domain of discourse that comprises:

(1) A set U of *users*.
(2) A set O of *objects*.
(3) A set A of *access privileges*.
(4) A set R of *roles*.

The U, O, A and R sets, comprise the (disjoint and finite) user, object, access privilege and role identifiers that form part of the universe of discourse for an RBAC program. More formally, the sets of constant symbols of interest in RBAC programs are:

—A countable set U of *user identifiers* such that $U = \mathcal{S} \cup \mathcal{J}$ where $\mathcal{S}$ is a set of character strings that identify users and $\mathcal{J} = \{u_j : j \in \mathbb{N} - \mid \mathcal{S} \mid\}$ where $\mathbb{N}$ is the set of natural numbers.
—A countable set O of *object identifiers* such that $O = \{o_k : k \in \mathbb{N}\}$.
—A countable set A of *access privileges* where $A = \{a_l : l \in \mathbb{N}\}$.[7]
—A countable set R of *role identifiers* such that $R = \{r_i : i \in \mathbb{N}\}$.

In this framework we have the following notions.

Definition 3.1. *An authorization is a triple (u_i, a_l, o_j) which denotes that a user u_i ($u_i \in U$) has the a_l access privilege ($a_l \in A$) on the object o_j ($o_j \in O$).*

The set of all authorizations defined by an RBAC program is represented by a set $AUTHORIZATION$.

Definition 3.2. *$AUTHORIZATION$ is a ternary relation, $AUTHORIZATION \subseteq U \times A \times O$.*

Definition 3.3. *If a_l is an access privilege and o_i is an object then a permission, p, is a pair (a_l, o_i) which denotes that a_l access is permitted on o_i.*

Definition 3.4. *$PERMISSION$ is a relation, $PERMISSION \subseteq A \times O$.*

Definition 3.5. *If a_l is an access privilege and o_i is an object then a denial, d, is a pair (a_l, o_i) which denotes that a_l access is denied on o_i.*

Definition 3.6. *$DENIAL$ is a relation, $DENIAL \subseteq A \times O$.*

Definition 3.7. *A permission-role assignment is a triple (a_l, o_i, r_j) which denotes that the permission (a_l, o_i) is assigned to the role r_j.*

Definition 3.8. *A denial-role assignment is a triple (a_l, o_i, r_j) which denotes that the denial (a_l, o_i) is assigned to the role r_j.*

A relation *URA* represents the set of user-role assignments that are defined by an RBAC program (*URA* is shorthand for user-role assignment).

[7]In practice, the principal forms of access privilege that are likely to be used are *read* and *write*.

DEFINITION 3.9. URA *is a relation,* URA $\subseteq U \times R$.

Similarly, we use a relation PRA to represent the set of permissions that are assigned to roles in a specification of an RBAC program (PRA is shorthand for permission-role assignment).

DEFINITION 3.10. PRA *is a ternary relation,* PRA $\subseteq A \times O \times R$.

For RBAC programs that may include denial-role assignments, we introduce a relation DRA that denotes the set of denials assigned to roles in a specification of an RBAC program (DRA is shorthand for denial-role assignment).

DEFINITION 3.11. DRA *is a ternary relation,* DRA $\subseteq A \times O \times R$.

To represent the URA, PRA and DRA relations in an RBAC constraint program Π, we respectively require the predicates ura, pra and dra with the intended meanings:

(1) $\Pi^\star \models_3 ura(u_i, r_j)$ *iff* $\langle u_i, r_j \rangle \in URA$.
(2) $\Pi^\star \models_3 pra(a_l, o_i, r_j)$ *iff* $\langle a_l, o_i, r_j \rangle \in PRA$.
(3) $\Pi^\star \models_3 dra(a_l, o_i, r_j)$ *iff* $\langle a_l, o_i, r_j \rangle \in DRA$.

In an RBAC program, the assignment of a user u_i to a role r_j is represented by a SA including definitions of the 2-place predicate, $ura(u_i, r_j)$. The assignment of a_l access on object o_i (the permission (a_l, o_i)) to a role r_j is represented in an RBAC program by a SA including definitions of the 3-place predicate, $pra(a_l, o_i, r_j)$. The denial of a_l access on object o_i (the denial (a_l, o_i)) to a role r_j is represented in an RBAC program by a SA including definitions of the 3-place predicate, $dra(a_l, o_i, r_j)$.

In the sequel, we will be interested in categories of RBAC programs that differ in the permission-role and denial-role assignments that they may include. The following notation will be used:

—An $RBAC_{H2A}$, $RBAC_{C3A}$ or $RBAC_{S4A}$ program Π that permits the expression of permission-role assignments but not denial-role assignments is denoted by $RBAC^P_{H2A}(\Pi)$, $RBAC^P_{C3A}(\Pi)$ and $RBAC^P_{S4A}(\Pi)$, respectively. Here, the superscript 'P' is shorthand for "permission".[8]

—An $RBAC_{H2A}$, $RBAC_{C3A}$ or $RBAC_{S4A}$ program Π that permits the expression of denial-role assignments but not permission-role assignments is denoted by $RBAC^D_{H2A}(\Pi)$, $RBAC^D_{C3A}(\Pi)$, and $RBAC^D_{S4A}(\Pi)$, respectively. In these cases, 'D' is shorthand for "denial".

—An $RBAC_{H2A}$, $RBAC_{C3A}$ or $RBAC_{S4A}$ program Π that permits the expression of both permission-role and denial-role assignments is denoted by $RBAC^{P,D}_{H2A}(\Pi)$, $RBAC^{P,D}_{C3A}(\Pi)$, and $RBAC^{P,D}_{S4A}(\Pi)$, respectively. In these cases, 'P, D' is used to specify that "permission and denial" role assignments may be defined in Π.

Temporal versions of $RBAC_{H2A}$, $RBAC_{C3A}$ and $RBAC_{S4A}$ will be respectively denoted by $TRBAC_{H2A}$, $TRBAC_{C3A}$ and $TRBAC_{S4A}$ with the superscripts P, D and the pair (P, D) being, respectively, used to denote versions of these programs

[8]The $RBAC^P_{H2A}$, $RBAC^P_{C3A}$ and $RBAC^P_{S4A}$ programs define the $RBAC_{H2A}$, $RBAC_{C3A}$ and $RBAC_{S4A}$ models in the SFK taxonomy [Sandhu et al. 2000].

that may include permission-role assignments only, denial-role assignments only, and both permission-role and denial-role assignments.

3.3 Role Hierarchies

In our proposal, an $RBAC_{H2A}$ role hierarchy is represented in an RBAC program that defines a $SENIOR\_TO$ relation ($SENIOR\_TO \subseteq R \times R$) as the reflexive-transitive closure of an irreflexive-intransitive DS relation ($DS \subseteq R \times R$); DS is short for "directly senior".

The $SENIOR\_TO$ relation is all pairs of roles $\langle r_i, r_j \rangle$ such that r_i $SENIOR\_TO$ r_j holds in the partial order $(R, SENIOR\_TO)$. The DS relation comprises all pairs of roles $\langle r_i, r_j \rangle$ $(r_i \neq r_j)$ such that $\langle r_i, r_j \rangle \in SENIOR\_TO$ and there is no role r_k $(r_i \neq r_k, r_j \neq r_k)$ such that $\langle r_i, r_k \rangle \in SENIOR\_TO$ and $\langle r_k, r_j \rangle \in SENIOR\_TO$.

The $SENIOR\_TO$ relation is represented in an $RBAC_{H2A}$ program Π by using a 2-place predicate *senior_to* with the intended interpretation:

—$\Pi^\star \models_3 senior\_to(r_i, r_j)$ *iff* $\langle r_i, r_j \rangle \in SENIOR\_TO$.

The DS relation is represented in an $RBAC_{H2A}$ program Π by using a 2-place predicate *ds* with the intended interpretation:

—$\Pi^\star \models_3 ds(r_i, r_j)$ *iff* $\langle r_i, r_j \rangle \in DS$.

The *ds* predicate may be defined in terms of *senior_to* thus (where r_i, r_j and r_k are distinct arbitrary roles):

$$\forall r_i, r_j \; [ds(r_i, r_j) \leftrightarrow senior\_to(r_i, r_j) \; \wedge$$
$$\neg(\exists r_k [senior\_to(r_i, r_k) \wedge senior\_to(r_k, r_j)])].$$

In the sequel, the notions of a *role path* and a *role inheritance path* will be important.

Definition 3.12. *If r_i and r_n are arbitrary roles in an $RBAC_{H2A}$ role hierarchy $\Re$ defined by an $RBAC_{H2A}$ program Π, and $\Pi^\star \models_3 ds(r_i, r_j) \wedge ds(r_j, r_k) \wedge \ldots \wedge ds(r_l, r_m) \wedge ds(r_m, r_n)$ $(i \neq j)$ then a role path, from r_i to r_n, exists in $\Re$ and is defined by the finite sequence:*

$$\langle r_i, r_j, r_k, \ldots, r_l, r_m, r_n \rangle.$$

Definition 3.13. *Given a role path, $RP = \langle r_i, r_j, \ldots, r_m, r_n \rangle$, the first element r_i of RP is referred to as the senior-most role in RP, and r_n is the junior-most role in RP.*

Definition 3.14. *If r_i and r_n are arbitrary roles in an $RBAC_{H2A}$ role hierarchy $\Re$ defined by an $RBAC_{H2A}$ program Π, and $\Pi^\star \models_3 senior\_to(r_i, r_j) \wedge senior\_to(r_j, r_k) \wedge \ldots \wedge senior\_to(r_l, r_m) \wedge senior\_to(r_m, r_n)$ then a role inheritance path, from r_i to r_n, exists in $\Re$ and is defined by the finite sequence:*

$$\langle r_i, r_j, r_k, \ldots, r_l, r_m, r_n \rangle.$$

Remark 3.15. *The $i \neq j$ condition in the definition of a role path does not apply to a role inheritance path. For the latter, i may or may not equal j (as senior_to, unlike ds, is reflexive).*

DEFINITION 3.16. *Given a role inheritance path, $RIP = \langle r_i, r_j, \ldots, r_m, r_n \rangle$, the first element r_i of RIP is referred to as the senior-most role in RIP, and r_n is the junior-most role in RIP.*

DEFINITION 3.17. *Given a finite sequence $\langle r_i, r_j, r_k, \ldots, r_l, r_m, r_n \rangle$ of role identifiers in a role path or role inheritance path, the index of a role $r \in \mathcal{R}$ where $\mathcal{R} = \{r_i, r_j, r_k, \ldots, r_l, r_m, r_n\}$, written $\mathcal{I}(r)$, is defined by the function $\mathcal{I} : \mathcal{R} \mapsto \mathcal{N}$ where $\mathcal{N} = \{1, .., p-1\}$, $p \in \mathbb{N}$, $\mathbb{N}$ is the set of natural numbers and $\mathcal{I}(r_i) \mapsto 1$, $\mathcal{I}(r_j) \mapsto 2$, $\ldots$, $\mathcal{I}(r_n) \mapsto \mid \mathcal{R} \mid$.*

In later sections, we will refer to sets of roles that inherit permission-role and denial-role assignments via a role inheritance path. For that we use the notions of a *down-role set* and an *up-role set*.

DEFINITION 3.18. *Let r ($r \in \mathcal{R}$) be a role in a role inheritance path $\mathcal{RIP}$ such that r is an element in $\mathcal{RIP}$.*

$$\uparrow r = \{r_i \in \mathcal{RIP} : \mathcal{I}(r_i) \geq \mathcal{I}(r)\} \quad \textit{(Up-role set)} \qquad .$$

$$\downarrow r = \{r_j \in \mathcal{RIP} : \mathcal{I}(r_j) \leq \mathcal{I}(r)\} \quad \textit{(Down-role set).}$$

DEFINITION 3.19. *Given a (non-empty) role path or role inheritance path, $\mathcal{P} = \langle r_i, r_j, \ldots, r_m, r_n \rangle$, the path length $\mathcal{L}$ between two roles r_k and r_l ($r_i \geq r_k \geq r_l \geq r_n$) in $\mathcal{P}$ is expressed thus: $\mathcal{L} = \mathcal{I}(r_l) - \mathcal{I}(r_k)$.*

DEFINITION 3.20. *The DS relation defines the set of pairs of roles $\langle r_i, r_j \rangle$ such that r_i and r_j are in a role path of length 1 induced from an $RBAC_{H2A}$ role hierarchy.*

DEFINITION 3.21. *The $SENIOR\_TO$ relation defines the set of pairs of roles $\langle r_i, r_j \rangle$ such that r_i and r_j are in a role inheritance path induced from an $RBAC_{H2A}$ role hierarchy where the length $\mathcal{L}$ between r_i and r_j is such that $\mathcal{L} \geq 1$.*

The types of RBAC and TRBAC programs that we describe below also include the notion of a user *session* [Sandhu et al. 1996]. A user's session will involve the user *activating* a subset of the roles to which the user has been assigned in order to acquire the permissions that are necessary to perform some task.

DEFINITION 3.22. *If U is the set of users and R is the set of roles then the relation $ACTIVE \subseteq U \times R$ is the set of pairs $\langle u_i, r_j \rangle$ such that user u_i is currently active in role r_j.*

In an RBAC program Π, a 2-place predicate $active(U, R)$, is used to specify that a user U is currently active in role R, and has the intended interpretation:

—$\Pi^\star \models_3 active(u_i, r_j)$ *iff* $\langle u_i, r_j \rangle \in ACTIVE$.

DEFINITION 3.23. *If a user u_m is active in the set of roles $\{r_1, r_2, \ldots, r_n\}$, and p_i is the set of permissions assigned to r_i via an RBAC program Π, for $i = (1, .., n)$, and there is no dra triple in Π that specifies that the permission p_i is denied to r_i then u_m has the set of permissions, $p_1 \cup p_2 \cup \cdots \cup p_n$.*

Whenever a user need not have a role activated, the user may *deactivate* the role. The following notions also apply to *RBAC* programs.

Definition 3.24. *In RBAC programs, the ura, pra, dra, ds, senior_to, and active predicates are called primitive predicates. All other predicates are called non-primitive.*

Remark 3.25. *The authorization triples $\langle u_i, a_j, o_k \rangle$ that are defined by RBAC programs (without constrained inheritance – see below) are expressed in terms of the primitive predicates.*

Definition 3.26. *The rule (rules) with the head $\alpha(U, A, O)$ (where α is an arbitrary predicate symbol) that define the authorization triples $\langle u_i, a_j, o_k \rangle$ for a class $\mathcal{C}$ of RBAC programs is (are) called the authorizations rule(s) for the $\mathcal{C}$ programs.*

4. REPRESENTING $RBAC^P_{H2A}$ PROGRAMS

The access control programs that we describe in this section are based on extended forms of the specifications of $RBAC^P_{H2A}$ as a logic program from [Barker 2000a] and [Barker 2001]. The extensions enable access control constraints to be defined in terms of comparison operators from the set $\{<, \leq, >, \geq, =, \neq\}$.

4.1 User-Role and Permission-Role Assignments by $RBAC^P_{H2A}$ Programs

In an $RBAC^P_{H2A}$ program, a user U is specified as being assigned to a role R by using definitions of the 2-place $ura(U, R)$ predicate.

Example 1. A SA may use the following ura rule to represent that "all non-probationary users, who are assigned to an area other than area 2 and who work on level 3 are assigned to the role r_9":

$$ura(U, r_9) \leftarrow area(U, A), level(U, L), not\ probationer(U), A \neq 2, L = 3.$$

□

In an $RBAC^P_{H2A}$ program, the assignment of A access on object O to a role R is expressed by using definitions of the 3-place $pra(A, O, R)$ predicate in the program. The type of object to be protected is unimportant in our formulation.

4.2 Representing $RBAC_{H2A}$ Role Hierarchies in RBAC Programs

In clause form logic, the definition of *senior_to* as the reflexive-transitive closure of ds may be expressed by the following (definite) program (in which '_' is an anonymous variable):

$$\begin{aligned}
&senior_to(R1, R1) \leftarrow ds(R1, _).\\
&senior_to(R1, R1) \leftarrow ds(_, R1).\\
&senior_to(R1, R2) \leftarrow ds(R1, R2).\\
&senior_to(R1, R2) \leftarrow ds(R1, R3), senior_to(R3, R2).
\end{aligned}$$

The clauses with the anonymous variable capture the reflexivity of $SENIOR\_TO$ by defining the identity relation ι for a set of roles R to wit: $\iota_R = \{(r_i, r_i) \mid r_i \in R\}$. The clause with the body $ds(R1, R2)$ defines DS. Hence, together these three clauses define the reflexive closure of DS, DS^ρ. That is: $DS^\rho = DS \cup \iota_R$. The pair of clauses with the head $senior\_to(R1, R2)$ defines the set of pairs $\langle r_i, r_j \rangle$ $(r_i, r_j \in R)$ in the transitive closure of DS, DS^τ, where $DS^\tau = DS^1 \cup DS^2 \cup \ldots \cup DS^{n-1} \cup DS^n$ and $i \in \{1, .., n\}$ is a *power* of DS.

Hence, $DS^\tau = \bigcup_{i=1}^{n} DS^i$. Moreover, we have $SENIOR\_TO = DS^\rho \cup DS^\tau$ and $SENIOR\_TO \cap SENIOR\_TO^{-1} \subseteq \iota_R$ (i.e., $SENIOR\_TO$ is anti-symmetric). Notice that the definition of $SENIOR\_TO$ assumes that the following first-order sentence is satisfied by an $RBAC$ program: $\forall r_i \in R\, \exists r_j \in R\, [ds(r_i, r_j) \vee ds(r_j, r_i)]$. That is, there is no role $r_k \in R$ in an $RBAC_{H2A}$ role hierarchy such that r_k is not related under DS to any role r_l ($r_k \neq r_l$) i.e., there are no "isolated" roles in an $RBAC_{H2A}$ role hierarchy and a single role is not a role hierarchy.[9]

Although *senior_to* is defined recursively, all goals terminate for this program because DS is acyclic. The acyclicity of DS is implied by the fact that DS is an irreflexive and intransitive relation. For implementations of $RBAC_{H2A}$ programs, we envisage representing *senior_to* as a set of facts. In this case, *senior_to* is clearly not recursive.

4.3 Representing Sessions in $RBAC^P_{H2A}$ Programs

The details of the implementation of a session management system to be used with $RBAC^P_{H2A}$ programs will be highly system-specific. In practice, we envisage a user u_i requesting to be active in a role r_j by appending an $activate(u_i, r_j)$ fact to an $RBAC^P_{H2A}$ program via a GUI. An $activate(u_i, r_j)$ fact represents the request by u_i to be active in role r_j. A set of rules of the form $active(U, R) \leftarrow activate(U, R), C$ will be included in the $RBAC^P_{H2A}$ program. An $active(u_i, r_j)$ fact will be implicitly appended to an $RBAC^P_{H2A}$ program if the conjunction of conditions, C, is true at the time of u_i's request to be active in r_j.

A user U deactivates a role $R1$ by retracting the appropriate *activate* fact from the $RBAC^P_{H2A}$ program. We assume that the set of roles a user has active are automatically deactivated when the user logs off the system.

4.4 User-role Review and $RBAC^P_{H2A}$ Programs

A SA may evaluate user-role review queries on an $RBAC^P_{H2A}$ program in order to extract information about the access policy the program represents.

DEFINITION 4.1. *A query $\mathcal{Q}$ is a user-role review query iff at least one literal in $\mathcal{Q}$ includes the ura predicate symbol.*

EXAMPLE 2. To determine which users are assigned to both of roles r_2 and r_3, a SA may evaluate the following user-role review query on an $RBAC^P_{H2A}$ program:

$$? - ura(U, r_2), ura(U, r_3).$$

□

Any number of user-role review queries may be expressed in a similar way to that illustrated in the previous example.

[9] If there were an isolated role r_k in an $RBAC_{H2A}$ role hierarchy then $ds(r_k, r_k)$ would not hold and so $senior\_to(r_k, r_k)$ would not hold.

4.5 Defining Authorized Access by $RBAC^P_{H2A}$ Programs

The set of authorization triples $\langle u_k, a_l, o_m \rangle$ defined by an $RBAC^P_{H2A}$ program may be expressed by the following first-order sentence (with sorts):[10]

$$\forall u_k \in U, \forall a_l \in A, \forall o_m \in O\ [\exists r_i, r_j \in R\ [ura(u_k, r_i)\ \wedge\ active(u_k, r_i)\ \wedge \\ senior\_to(r_i, r_j)\ \wedge\ pra(a_l, o_m, r_j)] \Rightarrow permitted(u_k, a_l, o_m)].$$

DEFINITION 4.2. *The authorizations rule for $RBAC^P_{H2A}$ programs is:*

$$permitted(U, A, O) \leftarrow ura(U, R1), active(U, R1), senior\_to(R1, R2), pra(A, O, R2).$$

The *permitted* rule expresses that a user U has A access on object O if: U is assigned to the role $R1$, U is active in $R1$, $R1$ is senior to a role $R2$ in an $RBAC_{H2A}$ role hierarchy, and $R2$ has been assigned the A access privilege on O.

Our formulation of *permitted* assumes that the activation by U of $R1$ also activates all roles that are junior to $R1$ in an $RBAC_{H2A}$ role hierarchy. However, other activation policies may be flexibly specified.[11] It should also be noted that the *permitted* rule defines the semantics of authorizations that is implicit (but not formally stated) in the $RBAC_{H2A}$ model described in [Sandhu et al. 2000]. In addition to defining the semantics of authorizations in an $RBAC^P_{H2A}$ program, we define an $RBAC^P_{H2A}$ program Π, written $RBAC^P_{H2A}(\Pi)$, in the following way.

DEFINITION 4.3. *Let $\mathcal{UAC}^P_{H2A}$ be the finite set of ura clauses in $RBAC^P_{H2A}(\Pi)$, let $\mathcal{PAC}^P_{H2A}$ be the finite set of pra clauses in $RBAC^P_{H2A}(\Pi)$, let $\mathcal{DSC}$ be the finite set of ds facts in $RBAC^P_{H2A}(\Pi)$, let $\mathcal{STC}$ be the set of clauses defining senior_to, let $\mathcal{AC}$ be the finite set of active rules and activate facts in $RBAC^P_{H2A}(\Pi)$, and let $\mathcal{PERMITTED}$ be the singleton set that includes the clause defining permitted. Then, $RBAC^P_{H2A}(\Pi)$ is defined in terms of the 6-tuple:*

$$(\mathcal{UAC}^P_{H2A}, \mathcal{PAC}^P_{H2A}, \mathcal{DSC}, \mathcal{STC}, \mathcal{AC}, \mathcal{PERMITTED}).$$

5. BEYOND $RBAC^P_{H2A}$ PROGRAMS

In this section, we show how some extensions to the $RBAC_{H2A}$ models from [Sandhu et al. 2000] may be represented by using RBAC programs. We begin by considering the representation of denial-role assignments.

5.1 Denial-Role Assignments in $RBAC^D_{H2A}$ and $RBAC^{P,D}_{H2A}$ Programs

An $RBAC^D_{H2A}$ program will include definitions of the 3-place $dra(A, O, R)$ predicate. An $RBAC^{P,D}_{H2A}$ program may include definitions of both *dra* and *pra* predicates. In the case of $RBAC^{P,D}_{H2A}$ programs, permission inheritance via an $RBAC_{H2A}$ role hierarchy may be affected by *overriding.*

DEFINITION 5.1. *In an $RBAC^{P,D}_{H2A}$ program Π, a denial (a_i, o_i) assigned to the role r_i overrides the permission (a_j, o_j) assigned to the role r_j iff $a_i = a_j$, $o_i = o_j$, $\Pi^\star \models_3 senior\_to(r_i, r_j)$, and if $\Pi^\star \models_3 dra(a_i, o_i, r_i)$ then $\Pi^\star \not\models_3 pra(a_j, o_j, r_j)$.*

---

[10] Although the clause form representation of *senior_to* is not first-order representable, *senior_to* may be equivalently expressed as a conjunction of atomic propositions.

[11] For instance, an $active(U, R2)$ condition may be added to the $permitted(U, A, O)$ rule to require that U be active in $R2$ in order to have A access on O.

DEFINITION 5.2. *If a permission (a_l, o_m) is specified in an $RBAC_{H2A}^{P,D}$ program Π as being assigned to a role r_j then a role r_i inherits the permission (a_l, o_m) from r_j iff $\Pi^\star \models_3$ senior_to(r_i, r_j) and there does not exist a denial (a_l, o_m) for r_i that overrides the permission (a_l, o_m).*

In the absence of overriding denials, the semantics of permission inheritance via an $RBAC_{H2A}$ role hierarchy may be expressed thus:

$$\forall a_l \in A,\ \forall o_k \in O,\ \forall r_i \in R\ [\exists r_j \in R\ [pra(a_l, o_k, r_j)\ \wedge\ senior\_to(r_i, r_j)] \Rightarrow pra(a_l, o_k, r_i)].$$

DEFINITION 5.3. *In an $RBAC_{H2A}^{P,D}$ program, Π, a permission (a_i, o_i) assigned to the role r_j overrides the denial (a_i, o_i) assigned to the role r_i iff $a_i = a_j$, $o_i = o_j$, $\Pi^\star \models_3$ senior_to(r_i, r_j), and if $\Pi^\star \models_3 pra(a_j, o_j, r_j)$ then $\Pi^\star \not\models_3 dra(a_i, o_i, r_i)$.*

DEFINITION 5.4. *If a denial (a_l, o_k) is specified in an $RBAC_{H2A}^{P,D}$ program Π as being assigned to a role r_i then a role r_j inherits the denial (a_l, o_k) from r_i iff $\Pi^\star \models_3$ senior_to(r_i, r_j) and there does not exist a permission (a_l, o_k) for r_j that overrides the denial (a_l, o_k).*

In the absence of overriding permissions, the semantics of denial inheritance via an $RBAC_{H2A}$ role hierarchy may be expressed thus:

$$\forall a_l \in A,\ \forall o_k \in O,\ \forall r_j \in R\ [\exists r_i \in R\ [dra(a_l, o_k, r_i)\ \wedge\ senior\_to(r_i, r_j)] \Rightarrow dra(a_l, o_k, r_j)].$$

5.2 Flexible Representation of Hierarchies in $RBAC_{H2A}$ Programs

It is often useful in practice for a SA to be able to specify more than just a role hierarchy (i.e., object, user, access privilege and group hierarchies may need to be represented). In this section, we consider the formulation of object hierarchies.

An object hierarchy may be used by a SA to express that if an object o_j is a subpart of an object o_i then A access on o_i implies A access on o_j. Conversely, a denial of A access on o_i implies the denial of A access on o_j. The following definitions of an *object instance* and a *subpart* apply.

DEFINITION 5.5. *Given a set of objects $\mathcal{O} \subseteq O$, we call each object $o_i \in \mathcal{O}$ an object instance of $\mathcal{O}$.*

DEFINITION 5.6. *An object o_j is a subpart of an object o_i if and only if every object instance in o_j is an object instance in o_i.*

A partial order $(O, INCLUDES)$ may be used to represent an $RBAC_{H2A}$ object hierarchy. The $INCLUDES$ relation ($INCLUDES \subseteq O \times O$) is the reflexive-transitive closure of an irreflexive-intransitive DI relation ($DI \subseteq O \times O$); DI is short for "directly includes".

The $INCLUDES$ relation comprises all pairs of objects $\langle o_i, o_j \rangle$ such that o_j is a sub-part of o_i. The DI relation comprises all pairs of objects $\langle o_i, o_j \rangle$ ($o_i \neq o_j$) such that $\langle o_i, o_j \rangle \in INCLUDES$ and there is no object o_k ($o_i \neq o_k, o_j \neq o_k$) such that $\langle o_i, o_k \rangle \in INCLUDES$ and $\langle o_k, o_j \rangle \in INCLUDES$.

The DI relation is represented in an $RBAC_{H2A}$ program Π by using a 2-place predicate di with the intended meaning:

—$\Pi^\star \models_3 di(o_i, o_j)$ *iff* $\langle o_i, o_j \rangle \in DI$.

The $INCLUDES$ relation is represented in an $RBAC_{H2A}$ program Π by using a 2-place predicate *includes* with the intended meaning:

—$\Pi^\star \models_3 includes(o_i, o_j)$ *iff* $\langle o_i, o_j \rangle \in INCLUDES$.

The di relation may be defined in terms of *includes* thus (where o_i, o_j and o_k are distinct arbitrary objects):

$$\forall o_i, o_j\ [di(o_i, o_j) \leftrightarrow includes(o_i, o_j)\ \wedge \\ \neg(\exists o_k[includes(o_i, o_k) \wedge includes(o_k, o_j)])].$$

The (definite) program that follows (in which '_' is an anonymous variable) defines *includes* as the reflexive-transitive closure of the di relation:

$$\begin{aligned}
&includes(O1, O1) \leftarrow di(O1, _).\\
&includes(O1, O1) \leftarrow di(_, O1).\\
&includes(O1, O2) \leftarrow di(O1, O2).\\
&includes(O1, O2) \leftarrow di(O1, O3), includes(O3, O2).
\end{aligned}$$

It should be noted that:

—all of the formal definitions relating to an $RBAC_{H2A}$ role hierarchy (see Section 4) extend (*mutatis mutandis*) to apply to RBAC object hierarchies.

—as with $RBAC_{H2A}$ role hierarchies, goal evaluation with respect to *includes* is guaranteed to be terminating because DI is acyclic.

—as with $RBAC_{H2A}$ role hierarchies, rather than representing an RBAC object hierarchy intensionally, an object hierarchy may be explicitly represented as a set of facts of the form $includes(o_i, o_j)$ that specify the set of all pairs $\langle o_i, o_j \rangle$ such that the object o_j is a subpart of the object o_i.

—any number of other hierarchies (e.g., access privilege hierarchies, subject hierarchies and group hierarchies) can be defined in a similar way to our representation of object hierarchies.

The *includes* predicate is used in the following variant of the *permitted* clause, from Section 4, to define the set of authorizations defined by an $RBAC^P_{H2A}$ program.

DEFINITION 5.7. *The authorizations rule for the class of $RBAC^P_{H2A}$ programs with RBAC object hierarchies is:*

$$permitted(U, A, O2) \leftarrow ura(U, R1), active(U, R1), senior\_to(R1, R2), \\ pra(A, O1, R2), includes(O1, O2).$$

This version of *permitted* specifies that: user U has A access on object $O2$ if U is assigned to, and is active in, a role $R1$ that is senior to a role $R2$, in an $RBAC_{H2A}$ role hierarchy, the A access privilege on an object $O1$ has been assigned to $R2$, and $O1$ includes $O2$ in an RBAC object hierarchy.

5.3 Flexible Specification of Inheritance Policies in $RBAC_{H2A}$ Programs

It is sometimes assumed that the roles in an organization may be represented by a single $RBAC_{H2A}$ role hierarchy [Sandhu et al. 1997]. However, this assumption may mean that, unless constraints are imposed, senior roles inherit more permissions than they strictly need in order to perform tasks (thus potentially violating the *Principle of Least Privilege* [Castano et al. 1995]).

In [Jajodia et al. 2001] and [Barker 2000b], the use of several (disjoint) role hierarchies is proposed with inheritance restricted to a particular sub-hierarchy. However, this approach can result in a proliferation of role hierarchies, and hence complex specifications of access policies.

In the approach that we propose, a global hierarchy of roles is defined in an $RBAC_{H2A}$ program, and a SA includes rules in the program, as required, to define the scope of the inheritance of permissions and denials. As we always use a global role hierarchy in our RBAC programs, it follows that for every $RBAC_{H2A}$ program Π the global $RBAC_{H2A}$ role hierarchy defined in Π will be *connected*.[12]

To describe flexible permission/denial inheritance policies formally, we introduce the notions of a *permission inheritance path* and a *denial inheritance path*.

Definition 5.8. *If* $\Sigma = \langle r_i, \ldots, r_n \rangle$ *is a role inheritance path and* p *is a permission then the subsequence of* Σ, $\sigma = \langle r_k, \ldots, r_l \rangle$ $(r_i \geq r_k \geq r_l \geq r_n)$, *is a permission inheritance path with respect to* p *iff* p *is assigned to* r_l *and* $\forall r_m$ $(r_k \geq r_m \geq r_l)$, r_m *inherits the* p *permission from* r_l.

Remark 5.9. *In terms of the notion of an up-role, given a permission inheritance path* $\langle r_k, \ldots, r_l \rangle$ *if* r_l *is assigned the* p *permission then* p *is inherited by all roles in the set* $\uparrow r_l$.

Definition 5.10. *If* $\Sigma = \langle r_i, \ldots, r_n \rangle$ *is a role inheritance path and* d *is a denial then the subsequence of* Σ, $\sigma = \langle r_k, \ldots, r_l \rangle$ $(r_i \geq r_k \geq r_l \geq r_n)$, *is a denial inheritance path with respect to* d *iff* d *is assigned to* r_k *and* $\forall r_m$ $(r_k \geq r_m \geq r_l)$, r_m *inherits the* d *denial from* r_k.

Remark 5.11. *In terms of the notion of a down-role, given a denial inheritance path* $\langle r_k, \ldots, r_l \rangle$ *if* r_k *is assigned the* d *denial then* d *is inherited by all roles in the set* $\downarrow r_k$.

To specify permission/denial inheritance via an RBAC program Π, a SA will use definitions of the *inherits_pra* and *inherits_dra* predicates with the intended meanings:

—$\Pi^\star \models_3 inherits\_pra(r_i, r_j, a_l, o_k)$ *iff* $\langle r_i, \ldots, r_j \rangle$ is a permission inheritance path and the a_l access privilege on object o_k is assigned to the role r_j.

—$\Pi^\star \models_3 inherits\_dra(r_i, r_j, a_l, o_k)$ *iff* $\langle r_i, \ldots, r_j \rangle$ is a denial inheritance path and the a_l access privilege on object o_k is denied to the role r_i.

Example 3. If $\Pi^\star \models_3 inherits\_pra(r_2, r_4, write, o_1)$ then every role on the permission inheritance path from role r_4 up to a role r_2 inherits the write access on o_1 assigned to r_4 if $\Pi^\star \models_3 pra(write, o_1, r_4)$. □

[12] Note that there are no isolated roles in an $RBAC_{H2A}$ role hierarchy so a single role cannot form a role hierarchy.

Definition 5.12. *The authorizations rule for an $RBAC^{P}_{H2A}$ program with restrictions on the upward inheritance of permission-role assignments is:*[13]

$$permitted(U, A, O) \leftarrow ura(U, R1), active(U, R1), senior\_to(R1, R2), pra(A, O, R2), inherits\_pra(R1, R2, A, O).$$

That is, a user U has A access on object O if: U is assigned to and is active in a role $R1$, $R1$ is senior to a role $R2$ in an $RBAC_{H2A}$ role hierarchy, and $R1$ can inherit the assignment of the A access privilege on O from $R2$.

5.4 Defining Authorized Access by $RBAC^{D}_{H2A}$ and $RBAC^{P,D}_{H2A}$ Programs

If denial-role assignments are specified via an $RBAC^{D}_{H2A}$ program Π, henceforth denoted $RBAC^{D}_{H2A}(\Pi)$, then an open policy may be represented. For instance, suppose that the semantics of an authorization $\langle u_i, a_l, o_k\rangle$ for an open policy specification is defined by the following first-order sentence (with sorts):

$$\forall u_i \in U, \forall a_l \in A, \forall o_k \in O\ [\exists r_i \in R\ [ura(u_i, r_i)\ \wedge\ active(u_i, r_i)\ \wedge \neg denied(r_i, a_l, o_k)] \Rightarrow\ access(u_i, a_l, o_k)].$$

That is, user u_i has a_l access on the object o_k if u_i is assigned to and is active in a role r_j that has not been denied the a_l access privilege on o_k.

The intended meaning of *denied* in the candidate open policy that we describe may be defined thus:[14]

$$\forall r_i \in R, \forall a_l \in A, \forall o_k \in O\ [\exists r_j \in R\ [senior\_to(r_j, r_i)\ \wedge\ dra(a_l, o_k, r_j)] \Rightarrow denied(r_i, a_l, o_k)].$$

Definition 5.13. *The authorizations rule for $RBAC^{D}_{H2A}$ programs is:*

$$access(U, A, O) \leftarrow ura(U, R1), active(U, R1), not\ denied(R1, A, O)$$

where denied is defined thus:

$$denied(R1, A, O) \leftarrow senior\_to(R3, R1), dra(A, O, R3).$$

The rule for *denied* specifies that a role $R1$ is denied the A access privilege on object O if a role $R3$ is senior to $R1$ in an $RBAC_{H2A}$ role hierarchy and $R3$ is denied A access on O. The $access(U, A, O)$ rule specifies that a user U has A access on O if there exists a role $R1$ to which U is assigned (and in which U is active), and there does not exist a role $R3$ that is senior to $R1$ in an $RBAC_{H2A}$ role hierarchy such that $R3$ (and thus U) has been denied A access on O.

Definition 5.14. *Let $\mathcal{UAC}^{D}_{H2A}$ be the finite set of ura clauses in $RBAC^{D}_{H2A}(\Pi)$, let $\mathcal{DAC}^{D}_{H2A}$ be the finite set of dra clauses in $RBAC^{D}_{H2A}(\Pi)$, let $\mathcal{DSC}$ be the finite set of ds facts in $RBAC^{D}_{H2A}(\Pi)$, let $\mathcal{STC}$ be the set of clauses that defines senior_to, let $\mathcal{AC}$ be the finite set of active rules and activate facts in $RBAC^{D}_{H2A}(\Pi)$, let*

[13] *Mutatis mutandis* the authorizations clauses for $RBAC^{D}_{H2A}$ and $RBAC^{P,D}_{H2A}$ programs with restricted inheritance of permissions and denials may be expressed in a similar way.

[14] Other open policies to the one that we consider are clearly possible. In particular, open policies in which a *denied*(u, a, o) predicate is defined, instead of *denied*(r_i, a, o), to specify denials of access to users rather than to roles.

$\mathcal{ACCESS}$ be the singleton set that includes the clause defining access, and let $\mathcal{DENIED}$ be the singleton set that includes the clause defining denied. Then, $RBAC^{D}_{H2A}(\Pi)$ is defined in terms of the 7-tuple:

$$(\mathcal{UAC}^{D}_{H2A}, \mathcal{DAC}^{D}_{H2A}, \mathcal{DSC}, \mathcal{STC}, \mathcal{AC}, \mathcal{ACCESS}, \mathcal{DENIED}).$$

If permission-role and denial-role assignments are included in an $RBAC^{P,D}_{H2A}$ program then various hybrid access policies may be represented, and various conflict resolution strategies may be defined to treat situations where conflicting permission and denial assignments are specified.

DEFINITION 5.15. *A pair of pra and dra facts, $pra(a_j, o_j, r_j)$ and $dra(a_i, o_i, r_i)$, is a conflicting assignment pair in an $RBAC^{P,D}_{H2A}$ program Π iff $\Pi^\star \models_3 senior\_to(r_i, r_j)$, $\Pi^\star \models_3 pra(a_j, o_j, r_j)$, $\Pi^\star \models_3 dra(a_i, o_i, r_i)$, $a_i = a_j$, and $o_i = o_j$.*

DEFINITION 5.16. *Suppose that a conflicting assignment pair exists in an $RBAC$ program Π such that $\Pi^\star \models_3 dra(a_i, o_i, r_i) \wedge pra(a_i, o_i, r_j) \wedge senior\_to(r_i, r_j)$. Moreover, let α denote an authorization expressed by $q(u_i, a_i, o_i)$ and suppose that u_i is assigned to and is active in role r_i. In this case, a conflict resolution strategy that ensures that $\Pi^\star \models_3 \neg\alpha$ is said to give preference to denials.*

DEFINITION 5.17. *Suppose that a conflicting assignment pair exists in an $RBAC$ program Π such that $\Pi^\star \models_3 dra(a_i, o_i, r_i) \wedge pra(a_i, o_i, r_j) \wedge senior\_to(r_i, r_j)$. Moreover, let α denote an authorization expressed by $q(u_i, a_i, o_i)$ and suppose that u_i is assigned to and is active in role r_i. In this case, a conflict resolution strategy that ensures that $\Pi^\star \models_3 \alpha$ is said to give preference to permissions.*

To represent a hybrid access policy, a SA needs to make only minor changes to the rules that are used to define authorizations in $RBAC^{P}_{H2A}$ or $RBAC^{D}_{H2A}$ programs, or simply needs to configure these rules in a different way.

DEFINITION 5.18. *The authorizations rule for an $RBAC^{P,D}_{H2A}$ program that implements a hybrid access policy in which denial-role assignments may be used to override conflicting permission-role assignments is:*

$$\begin{aligned} authorized(U, A, O) \leftarrow\ & ura(U, R1), active(U, R1), senior\_to(R1, R2), \\ & pra(A, O, R2), not\ denied(R1, A, O). \end{aligned}$$

That is, U has the A access privilege on O if U is assigned to and is active in a role $R1$, $R1$ is senior to a role $R2$ in an $RBAC_{H2A}$ role hierarchy, $R2$ has been assigned A access on O, and $R1$ is not specified as being denied A access on O.

Henceforth, we use $RBAC^{P,D}_{H2A}(\Pi_d)$ to denote an arbitrary $RBAC^{P,D}_{H2A}$ program Π_d that implements a "denial-role assignments have precedence" conflict resolution strategy.

DEFINITION 5.19. *Let $\mathcal{UAC}^{P,D}_{H2A}$ be the finite set of ura clauses in $RBAC^{P,D}_{H2A}(\Pi_d)$, let $\mathcal{PAC}^{P,D}_{H2A}$ be the finite set of pra clauses in $RBAC^{P,D}_{H2A}(\Pi_d)$, let $\mathcal{DAC}^{P,D}_{H2A}$ be the finite set of dra clauses in $RBAC^{P,D}_{H2A}(\Pi_d)$, let $\mathcal{DSC}$ be the finite set of ds facts in $RBAC^{P,D}_{H2A}(\Pi_d)$, let $\mathcal{STC}$ be the set of clauses defining senior_to, let $\mathcal{AC}$ be the finite set of active rules and activate facts in $RBAC^{P,D}_{H2A}(\Pi_d)$, let $\mathcal{AUTH}$ be the singleton that includes the clause defining authorized, and let $\mathcal{DENIED}$ be the singleton set* *that includes the clause defining denied. Then,* $RBAC^{P,D}_{H2A}(\Pi_d)$ *is defined in terms of the 8-tuple:*

$$(\mathcal{UAC}^{P,D}_{H2A}, \mathcal{PAC}^{P,D}_{H2A}, \mathcal{DAC}^{P,D}_{H2A}, \mathcal{DSC}, \mathcal{STC}, \mathcal{AC}, \mathcal{AUTH}, \mathcal{DENIED}).$$

THEOREM 5.20. *Suppose that a user* u_m *is assigned to and is active in a role* r_i *in an* $RBAC^{P,D}_{H2A}$ *program* Π_d *and* $\Pi^\star_d \models_3 senior\_to(r_k, r_i)$ *and* $\Pi^\star_d \models_3 senior\_to(r_i, r_j)$ *(where* $r_i, r_j, r_k \in R$*). Then,* $\Pi^\star_d \models_3 authorized(u_m, a_l, o_k)$ *iff* $\Pi^\star_d \models_3 pra(a_l, o_n, r_j)$ *and* $\Pi^\star_d \not\models_3 dra(a_l, o_n, r_k)$*.*

PROOF. For the only if direction, if $\Pi^\star_d \not\models_3 pra(a_l, o_n, r_j)$ then $\neg\exists\ r_l \in R$ such that $\Pi^\star_d \models_3 senior\_to(r_j, r_l)$ and $\Pi^\star_d \models_3 pra(a_l, o_n, r_l)$, but then $\Pi^\star_d \not\models_3 authorized(u_m, a_l, o_k)$. If $\Pi^\star_d \models_3 dra(a_l, o_n, r_k)$ and $\Pi^\star_d \models_3 senior\_to(r_k, r_i)$ then $\Pi^\star_d \models_3 denied(r_i, a_l, o_k)$ and so $\Pi^\star_d \not\models authorized(u_m, a_l, o_k)$.

For the if direction, we have, from the truth of the premise, that $\neg\exists r_k \in R$ such that $\Pi^\star_d \models_3 senior\_to(r_k, r_i)$ and $\Pi^\star_d \models_3 dra(a_l, o_n, r_k)$. Hence, we have $\Pi^\star_d \not\models_3 denied(r_i, a_l, o_k)$. Then, as u_m is assigned to and is active in r_i, $\Pi^\star_d \models_3 pra(a_l, o_n, r_j)$, and $\Pi^\star_d \models_3 senior\_to(r_i, r_j)$, we have $\Pi^\star_d \models_3 authorized(u_m, a_l, o_k)$. □

PROPOSITION 5.21. *If an* $RBAC^{P,D}_{H2A}$ *program* Π_d *includes a conflicting assignment pair,* $dra(a_l, o_n, r_i)$ *and* $pra(a_l, o_n, r_j)$*, and user* u_m *is assigned to and active in role* r_i *then* $\Pi^\star_d \not\models_3 authorized(u_m, a_l, o_k)$*.*

PROOF. If Π_d includes a conflicting assignment pair then we have $\Pi^\star_d \models_3 dra(a_l, o_n, r_i)$, $\Pi^\star_d \models_3 pra(a_l, o_n, r_j)$, and $\Pi^\star_d \models_3 senior\_to(r_i, r_j)$. As $\Pi^\star_d \models_3 senior\_to(r_i, r_i)$, it follows that $\Pi^\star_d \models_3 denied(r_i, a_l, o_k)$, and hence $\Pi^\star_d \not\models_3 authorized(u_m, a_l, o_k)$. □

COROLLARY 5.22. *Let* Π_d *be an* $RBAC^{P,D}_{H2A}$ *program. Then,* $\Pi^\star_d \not\models_3 authorized(u_m, a_l, o_k)$ *iff* $\Pi^\star_d \not\models_3 pra(a_l, o_n, r_j) \vee \Pi^\star_d \models_3 dra(a_l, o_n, r_k)$ *when* $\Pi^\star_d \models_3 senior\_to(r_k, r_i)$*,* $\Pi^\star_d \models_3 senior\_to(r_i, r_j)$*, and* a_m *is assigned to and is active in role* r_i*.*

PROOF. An immediate consequence of Theorem 5.20. □

The rules that we use to specify the $\langle u_m, a_l, o_k \rangle$ triples that are expressed by an $RBAC^{P}_{H2A}$, $RBAC^{D}_{H2A}$ or $RBAC^{P,D}_{H2A}$ program define the semantics of authorization that we propose for access control policies that are represented by these types of programs. Other candidate semantics (e.g., to accommodate multiple, disjoint role hierarchies) are possible by making suitable adjustments to our definitions.

The first-order sentence (with sorts) that follows defines the authorizations specified by an $RBAC^{P,D}_{H2A}$ program that implements a "permission-role assignments have precedence" conflict resolution strategy:

$$\begin{aligned}\forall u_m \in U, \forall a_l \in A, \forall o_k \in O\ [\exists r_i, r_j \in R\ [[ura(u_m, r_i)\ \wedge\ active(u_m, r_i) \wedge \\ senior\_to(r_i, r_j)\ \wedge\ pra(a_l, o_k, r_j)]\ \vee\ [ura(u_m, r_i)\ \wedge\ active(u_m, r_i) \wedge \\ \neg denied(r_i, a_l, o_k)]] \Rightarrow allowed(u_m, a_l, o_k)].\end{aligned}$$

Henceforth, we use $RBAC^{P,D}_{H2A}(\Pi_p)$ to denote an arbitrary $RBAC^{P,D}_{H2A}$ program Π_p that implements a "permission-role assignments have precedence" conflict resolution strategy.

DEFINITION 5.23. *The authorizations rules for* $RBAC^{P,D}_{H2A}(\Pi_p)$ *are:*

$$allowed(U,A,O) \leftarrow ura(U,R1), active(U,R1), senior\_to(R1,R2), pra(A,O,R2).$$
$$allowed(U,A,O) \leftarrow ura(U,R1), active(U,R1), not\ denied(R1,A,O).$$

That is, user U has the A access privilege on object O if U is assigned to, and is active in, a role $R1$ that inherits A access on O or if U is assigned to and is active in a role $R1$ that has not been denied A access on O. Moreover, permission-role assignments will override conflicting denial-role assignments, and imply that U has A access on O.

DEFINITION 5.24. *Let* $\mathcal{UAC}^{P,D}_{H2A}$ *be the finite set of ura clauses in* $RBAC^{P,D}_{H2A}(\Pi_p)$*, let* $\mathcal{PAC}^{P,D}_{H2A}$ *be the finite set of pra clauses in* $RBAC^{P,D}_{H2A}(\Pi_p)$*, let* $\mathcal{DAC}^{P,D}_{H2A}$ *be the finite set of dra clauses in* $RBAC^{P,D}_{H2A}(\Pi_p)$*, let* $\mathcal{DSC}$ *be the finite set of ds facts in* $RBAC^{P,D}_{H2A}(\Pi_p)$*, let* $\mathcal{STC}$ *be the set of clauses defining senior_to, let* $\mathcal{AC}$ *be the finite set of active rules and activate facts in* $RBAC^{P,D}_{H2A}(\Pi_p)$*, let* $\mathcal{ALLOWED}$ *be the set of clauses defining allowed, and let* $\mathcal{DENIED}$ *be the singleton set that includes the clause defining denied. Then,* $RBAC^{P,D}_{H2A}(\Pi_p)$ *is defined in terms of the 8-tuple:*

$$(\mathcal{UAC}^{P,D}_{H2A}, \mathcal{PAC}^{P,D}_{H2A}, \mathcal{DAC}^{P,D}_{H2A}, \mathcal{DSC}, \mathcal{STC}, \mathcal{AC}, \mathcal{ALLOWED}, \mathcal{DENIED}).$$

THEOREM 5.25. *Let* u_m *be a user that is assigned to and is active in a role* r_i *in an* $RBAC^{P,D}_{H2A}$ *program* Π_p *such that* $senior\_to(r_k, r_i)$ *and* $senior\_to(r_i, r_j)$ *hold (where* $r_i, r_j, r_k \in R$*). Then,* $\Pi^\star_p \not\models_3 allowed(u_m, a_l, o_n)$ *iff* $\Pi^\star_p \not\models_3 pra(a_l, o_n, r_j)$ *and* $\Pi^\star_p \models_3 dra(a_l, o_n, r_k)$*.*

PROOF. For the only if direction, by contra-position, there are two cases to consider: where $\Pi^\star_p \models_3 pra(a_l, o_n, r_j)$ and where $\Pi^\star_p \not\models_3 dra(a_l, o_n, r_k)$. If $\Pi^\star_p \models_3 pra(a_l, o_n, r_j)$ then, as $\Pi^\star_p \models_3 senior\_to(r_i, r_j)$ and u_m is assigned to and is active in r_i, we have $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$. If $\Pi^\star_p \not\models_3 dra(a_l, o_n, r_k)$ then, as $\Pi^\star_p \models_3 senior\_to(r_k, r_i)$, we have $\Pi^\star_p \models_3 \neg denied(r_i, a_l, o_n)$ and, as u_m is assigned to and is active in r_i, $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$.

For the if direction, suppose that $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$. Given that u_m is assigned to and is active in r_i and $\Pi^\star_p \models_3 senior\_to(r_i, r_j)$ and $\Pi^\star_p \models_3 senior\_to(r_k, r_i)$ then for $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$ we must have either $\Pi^\star_p \models_3 pra(a_l, o_n, r_j)$ or $\Pi^\star_p \not\models_3 dra(a_l, o_n, r_k)$. However, $\Pi^\star_p \not\models_3 pra(a_l, o_n, r_j)$ and $\Pi^\star_p \models_3 dra(a_l, o_n, r_k)$. □

PROPOSITION 5.26. *If an* $RBAC^{P,D}_{H2A}$ *program* Π_p *includes a conflicting assignment pair,* $dra(a_l, o_n, r_i)$ *and* $pra(a_l, o_n, r_j)$*, and user* u_m *is assigned to and is active in role* r_j *then* $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$*.*

PROOF. If Π_p includes a conflicting assignment pair then we have $\Pi^\star_p \models_3 dra(a_l, o_n, r_i)$, $\Pi^\star_p \models_3 pra(a_l, o_n, r_j)$, and $\Pi^\star_p \models_3 senior\_to(r_i, r_j)$. As $\Pi^\star_p \models_3 senior\_to(r_j, r_j)$ must always hold and $\Pi^\star_p \models_3 ura(u_m, r_j) \wedge active(u_m, r_j) \wedge pra(a_l, o_n, r_j)$ it follows that $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$. □

COROLLARY 5.27. $\Pi^\star_p \models_3 allowed(u_m, a_l, o_n)$ *iff* $\Pi^\star_p \models_3 pra(a_l, o_n, r_j) \vee \Pi^\star_p \not\models_3 dra(a_l, o_n, r_k)$ *where* $\Pi^\star_p \models_3 senior\_to(r_k, r_i)$*,* $\Pi^\star_p \models_3 senior\_to(r_i, r_j)$ *and* u_m *is assigned to and is active in role* r_i*.*

PROOF. An immediate consequence of Theorem 5.25. □

Program	Rules Defining Authorizations
$RBAC^{P}_{H2A}(\Pi)$	$permitted(U, A, O) \leftarrow ura(U, R1), active(U, R1), senior\_to(R1, R2), pra(A, O, R2)$
$RBAC^{D}_{H2A}(\Pi)$	$access(U, A, O) \leftarrow ura(U, R1), active(U, R1), not\ denied(R1, A, O)$ $denied(R1, A, O) \leftarrow senior\_to(R3, R1), dra(A, O, R3)$
$RBAC^{P,D}_{H2A}(\Pi_d)$	$authorized(U, A, O) \leftarrow ura(U, R1), active(U, R1), senior\_to(R1, R2), pra(A, O, R2), not\ denied(R1, A, O)$ $denied(R1, A, O) \leftarrow senior\_to(R3, R1), dra(A, O, R3)$
$RBAC^{P,D}_{H2A}(\Pi_p)$	$allowed(U, A, O) \leftarrow ura(U, R1), active(U, R1), senior\_to(R1, R2), pra(A, O, R2)$ $allowed(U, A, O) \leftarrow ura(U, R1), active(U, R1), not\ denied(R1, A, O)$

Table II. The rules defining the authorization triples $\langle u, a, o \rangle$ in $RBAC_{H2A}$ programs.

Predicate	Arity	Semantics
ura	2	$ura(U, R)$ specifies that user U is assigned to role R.
pra	3	$pra(A, O, R)$ specifies that role R is assigned A access on object O.
dra	3	$dra(A, O, R)$ specifies that role R is denied A access on object O.
ds	2	$ds(R1, R2)$ specifies that role $R1$ is directly senior to role $R2$
senior_to	2	$senior\_to(R1, R2)$ specifies that role $R1$ is directly or indirectly senior to role $R2$ in an $RBAC_{H2A}$ role hierarchy.
active	2	$active(U, R)$ specifies that user U is active in role R.

Table III. The primitive predicates used in $RBAC_{H2A}$ programs.

An alternative to allowing a conflicting pair of permission-role and denial-role assignments to coexist in an $RBAC^{P,D}_{H2A}$ program is to regard this as an error situation, and to impose constraints on an $RBAC^{P,D}_{H2A}$ program to prevent the occurrence of conflicting assignment pairs. Specifically, a constraint may be included in an $RBAC^{P,D}_{H2A}$ program to ensure that the condition $PRA \cap DRA = \emptyset$ is satisfied.

The clauses that we propose for defining the set of authorizations described by $RBAC^{P}_{H2A}$, $RBAC^{D}_{H2A}$ and $RBAC^{P,D}_{H2A}$ programs (with different conflict resolution strategies) are summarized in Table II. The semantics of the primitive predicates that we use to represent the different types of $RBAC_{H2A}$ programs that we have considered are summarized in Table III.

5.5 $RBAC_{H2A}$ Programs and the SFK Taxonomy

The relationship between $RBAC^{P}_{H2A}$ constraint programs and the $RBAC_{H2A}$ model from [Sandhu et al. 2000] is important to understand and may be described thus:

—The *ura* and *pra* predicates in $RBAC^{P}_{H2A}$ programs respectively enable many-to-many user-role and permission-role assignments to be expressed;

—user-role reviews are performed on $RBAC^{P}_{H2A}$ programs by a SA posing *ura* queries on the program;

—an $RBAC^{P}_{H2A}$ program only permits users to acquire permissions through roles;

—the *active* predicate enables users to be active in multiple roles simultaneously (and hence enables users to use permissions of multiple roles simultaneously);

—*senior_to* defines an unrestricted $RBAC_{H2A}$ role hierarchy as the reflexive-transitive closure of the irreflexive and intransitive *ds* relation.

For $RBAC^{D}_{H2A}$ programs, definitions of dra are used to specify denial-role assignments. Denial-role assignments are not part of the SFK models. In $RBAC^{P,D}_{H2A}$ programs, both pra and dra clauses may be defined. Hence, $RBAC^{P,D}_{H2A}$ programs extend the set of RBAC policies that may be represented beyond those that may be expressed in terms of the $RBAC_{H2A}$ model from [Sandhu et al. 2000].

5.6 Representing $RBAC^{P}_{C3A}$, $RBAC^{D}_{C3A}$ and $RBAC^{P,D}_{C3A}$ Programs

In this section, we consider the extension of $RBAC^{P}_{H2A}$, $RBAC^{D}_{H2A}$ and $RBAC^{P,D}_{H2A}$ programs to include separation of duties constraints [Sandhu et al. 2000]. That is, we consider the representation of $RBAC^{P}_{C3A}$, $RBAC^{D}_{C3A}$ and $RBAC^{P,D}_{C3A}$ programs. In the sequel, we will simply refer to $RBAC_{C3A}$ programs as the discussion that follows applies equally well to $RBAC^{P}_{C3A}$, $RBAC^{D}_{C3A}$ and $RBAC^{P,D}_{C3A}$ programs.

The *static separation of duties (ssd)* constraint is used to specify that a user cannot be assigned to a pair of mutually exclusive roles [Sandhu et al. 2000]. The *dynamic separation of duties (dsd)* constraint is used to prevent a user simultaneously activating a pair of roles that are specified as being dynamically separated [Sandhu et al. 2000].

A symmetric relation, $SSD \subseteq R \times R$, represents pairs of roles that are statically separated; a symmetric relation, $DSD \subseteq R \times R$, represents pairs of roles that are dynamically separated. In an $RBAC_{C3A}$ program Π, a 2-place ssd and a 2-place dsd predicate are used to represent pairs of roles (r_i, r_j) that are, respectively, statically and dynamically separated. These predicates have the intended meanings:

(1) $\Pi^{\star} \models_3 ssd(r_i, r_j)$ *iff* $\langle r_i, r_j \rangle \in SSD$.

(2) $\Pi^{\star} \models_3 dsd(r_i, r_j)$ *iff* $\langle r_i, r_j \rangle \in DSD$.

To specify two roles r_i and r_j as being statically separated in an $RBAC_{C3A}$ program, a SA may include an $ssd(r_i, r_j)$ fact and, because of the symmetry of the ssd relation, the fact $ssd(r_j,r_i)$ $(r_i \neq r_j)$. To specify a pair of roles r_i and r_j as being dynamically separated in an $RBAC_{C3A}$ program, a SA will include a $dsd(r_i, r_j)$ fact and, because of the symmetry of the dsd relation, a $dsd(r_j,r_i)$ fact $(r_j \neq r_i)$. Alternatively, the symmetric closure of the ssd (and dsd likewise) relations can be computed by using the program:[15]

$$sym\_ssd(R1, R2) \leftarrow ssd(R1, R2).$$
$$sym\_ssd(R1, R2) \leftarrow ssd(R2, R1).$$

The semantics of a violation of the ssd constraint may be specified in terms of the notion of a *state*.

DEFINITION 5.28. *The state of an $RBAC_{C3A}$ program Π at a time t is the set of rules in Π at time t.*

DEFINITION 5.29. *The ssd constraint is inconsistent with respect to a state of an $RBAC_{C3A}$ program Π iff there exists a user u_m $(u_m \in U)$ such that $\Pi^{\star} \models_3 ura(u_m, r_i)$, $\Pi^{\star} \models_3 ura(u_m, r_j)$, and $\Pi^{\star} \models_3 ssd(r_i, r_j)$ (where $r_i \in R$, $r_j \in R$ and $r_i \neq r_j$).*

[15] The sym_ssd and sym_dsd programs remove the burden of ensuring that ssd and dsd are symmetric from the security administrator.

In an $RBAC_{C3A}$ program, the *ssd* constraint may be represented by using the following rule:[16]

$$inconsistent\_ssd(U, R1) \leftarrow ura(U, R1), ssd(R1, R2), ura(U, R2).$$

The *inconsistent_ssd* constraint is used to express that the assignment of a user U to the role $R1$ is inconsistent with the *ssd* constraint if U is also specified in an $RBAC_{C3A}$ program as being assigned to the role $R2$, and $R1$ and $R2$ are specified as being a statically separated pair of roles.

The *ssd* constraint, is checked by a SA whenever a $ura(U, R1)$ rule is to be inserted into an $RBAC_{C3A}$ program. We describe the details of constraint checking in Section 8, after we have considered the operational method that is used to check constraints.

Definition 5.30. *The dsd constraint is inconsistent with respect to an* $RBAC_{C3A}$ *program* Π *iff there is a user* u_m $(u_m \in U)$ *such that* $\Pi^\star \models_3 active(u_m, r_i)$, $\Pi^\star \models_3 active(u_m, r_j)$, *and* $\Pi^\star \models_3 dsd(r_i, r_j)$ *(where* $r_i \in R$, $r_j \in R$ *and* $r_i \neq r_j$*).*

To represent the *dsd* constraint, the following pair of rules may be included in an $RBAC_{C3A}$ program:[17]

$$active(U, R1) \leftarrow activate(U, R1), not\ inconsistent\_dsd(U, R1).$$
$$inconsistent\_dsd(U, R1) \leftarrow dsd(R1, R2), activate(U, R2).$$

It follows that a user u_m is implicitly recorded as being active in r_i if u_m has not also requested to be active in a role r_j such that $\Pi^\star \models_3 dsd(r_i, r_j)$.

Definition 5.31. *An* $RBAC_{C3A}$ *program is an* $RBAC_{H2A}$ *program that is extended to include the definitions of inconsistent_ssd and inconsistent_dsd and sets of ssd and dsd facts.*

5.7 Representing $RBAC^{P}_{S4A}$, $RBAC^{D}_{S4A}$ and $RBAC^{P,D}_{S4A}$ Programs

To extend $RBAC_{C3A}$ programs to $RBAC_{S4A}$ programs, permission-role and denial-role reviews must be supported.

Definition 5.32. *A query* $\mathcal{Q}$ *is a permission-role review query iff at least one literal in* $\mathcal{Q}$ *includes the pra predicate symbol.*

Definition 5.33. *A query* $\mathcal{Q}$ *is a denial-role review query iff at least one literal in* $\mathcal{Q}$ *includes the dra predicate symbol.*

As in the case of user-role reviews, permission-role and denial-role reviews are straightforward to perform on $RBAC_{S4A}$ constraint programs; a SA may simply evaluate *pra* queries and *dra* queries with respect to an $RBAC_{S4A}$ program.

[16] Our definition of the $inconsistent\_ssd(U, R1)$ rule assumes that there is no role r_k such that $ssd(r_i, r_j)$, $senior\_to(r_k, r_i)$ and $senior\_to(r_k, r_j)$ hold in an $RBAC_{C3A}$ program. That is, there is no role that is senior to a pair of statically separated roles in a role hierarchy defined in an $RBAC_{C3A}$ program. We can use an integrity constraint to avoid this situation arising.

[17] We assume that there does not exist a role r_k such that $dsd(r_i, r_j)$, $senior\_to(r_k, r_i)$ and $senior\_to(r_k, r_j)$ hold in an $RBAC_{C3A}$ program.

Any number of rules may be seamlessly incorporated into $RBAC^{P}_{S4A}$, $RBAC^{D}_{S4A}$ or $RBAC^{P,D}_{S4A}$ programs to provide SAs with additional policy review options. Moreover, answers to policy review queries, that could not be generated by using logic programming, may often be expressed in terms of constraints (see below).

As $RBAC_{S4A}$ programs with permission-role and denial-role reviews do not strictly extend $RBAC_{C3A}$ programs, we choose to define $RBAC_{S4A}$ programs in the following way.

DEFINITION 5.34. *An $RBAC_{S4A}$ program is an $RBAC_{C3A}$ program that is extended to include rules for enabling an arbitrary number of SA review queries to be performed.*

6. FLEXIBLE TEMPORAL AUTHORIZATION POLICIES BY TRBAC PROGRAMS

In this section, we consider the extension of RBAC programs to include temporal authorizations: TRBAC (constraint) programs.

Temporal authorization models are increasingly being recognized as important in practice. Temporal authorizations enable a SA to specify that user access to a data item is to hold for a restricted interval of time, and automatically expires as soon as the maximum time point in the interval is reached. Temporal authorizations provide a SA with a fine level of control on assignments, they can limit the damage an unauthorized user may wreak if the user gains access to a system, they provide a revocation mechanism that is difficult to implement effectively in distributed systems, and they are important in workflow applications [Atluri and Huang 1996].

In the sequel, we consider the representation of $TRBAC^{P,D}_{H2A}$ programs with integer constraints used to represent temporal information and with denial-role assignments having precedence. The $TRBAC^{P}_{H2A}$ and $TRBAC^{D}_{H2A}$ programs are just special cases of the $TRBAC^{P,D}_{H2A}$ programs that we consider, and $TRBAC^{P,D}_{H2A}$ programs with permission-role assignments having precedence are just a variant of the $TRBAC^{P,D}_{H2A}$ programs that we describe.

Temporal constraints are expressed in a $TRBAC^{P,D}_{H2A}$ program with respect to user-role assignments, permission-role assignments, and denial-role assignments. The $TRBAC^{P,D}_{H2A}$ programs that we describe are a natural extension of the $RBAC^{P,D}_{H2A}$ programs. Hence, the predicates that we use for $TRBAC^{P,D}_{H2A}$ programs are essentially the same as those that we used for $RBAC^{P,D}_{H2A}$ programs. The only significant difference is that, in $TRBAC^{P,D}_{H2A}$ programs, an extra argument is added to the *ura*, *pra* and *dra* predicates that are defined in $RBAC^{P,D}_{H2A}$ programs to, respectively, represent the times at which user-role, permission-role, and denial-role assignments hold. It follows that the authorizations that are defined by the $TRBAC^{P,D}_{H2A}$ programs that we will discuss may be expressed by the following first-order sentence (with sorts and with T denoting a set of times):

$$\begin{aligned}\forall u_m \in U, \forall a_l \in A, \forall o_n \in O, \forall t_k \in T\ [\exists r_i, r_j \in R\ [ura(u_m, r_i, t_k) \wedge \\ active(u_m, r_i) \wedge senior\_to(r_i, r_j) \wedge pra(a_l, o_n, r_j, t_k) \wedge \\ \neg denied(r_i, a_l, o_n, t_k)] \Rightarrow authorized(u_m, a_l, o_n, t_k)]\end{aligned}$$

where *denied* is defined by the sentence:

$$\forall r_i \in R, \forall o_n \in O, \forall a_l \in A, \forall t_k \in T\ [\exists r_j \in R\ [senior\_to(r_j, r_i)\ \wedge \\ dra(r_j, a_l, o_n, t_k)] \Rightarrow denied(r_i, a_l, o_n, t_k)].$$

6.1 Representing Time in $TRBAC^{P,D}_{H2A}$ Programs

In our representation of $TRBAC$ programs, time is approximated by using discrete time points. These time points are consecutive, equally distanced and isomorphic to the natural numbers. For ease of presentation, we shall use integers of the form $y_1y_2y_3y_4m_1m_2d_1d_2$ to represent a time point: the d_1d_2th day of the m_1m_2th month of the $y_1y_2y_3y_4$th year. Thus, $y_1y_2y_3y_4$ are the four year digits, e.g. 2001, 1996, m_1m_2 are the month digits, e.g. 11 (November), 03 (March), and d_1d_2 are the day of the month digits, e.g. 05. For example, 20010716 represents the 16th of July 2001. Again for simplicity, we do not consider time below the granularity of single days; the extension to add hours, minutes and seconds is straightforward. In practice, a standard representation such as the Unix time integer (number of seconds since the Epoch (00:00:00 UTC, January 1, 1970) will be used).

We assume that a 1-place *date* predicate exists in the language that is used to obtain the current time point. In practice, the *date* predicate will extract the current time from the system clock.

For providing flexibility and ease of definition for a SA, we will assume some given constraint predicates that map a time point into various components:

—$year(T, Y)$ constrains Y to be the year of time point T.
—$month(T, M)$ constrains M to be the month of time point T.
—$day(T, D)$ constrains D to be the day of time point T.
—$weekday(T, W)$ constrains W to be the weekday of time point T, where 1 represents Monday, 2 represents Tuesday, ..., 7 represents Sunday.

Given our representation of time, it is straightforward to define the first three predicates by using integer constraints as follows:

$$year(T, Y) \leftarrow Y = T \div 10000.$$
$$month(T, M) \leftarrow M = (T\ mod\ 10000) \div 100, 01 \leq M, M \leq 12.$$
$$day(T, D) \leftarrow D = T\ mod\ 100, 01 \leq D, D \leq 31.$$

It is also possible to define the *weekday* predicate by using a conjunction of integer constraints, though not straightforward. In practice, we can use specialized code to make the calculation. In the sequel, we will use $\mathcal{M}$ to denote the set of constraint predicates that define a mapping of a time into its component parts.

Constraints may be imposed on the year, month, day or weekday individually using the *year*, *month*, *day* and *weekday* predicates. For example, the goal $weekday(T, W), W = 4$ requires that the day of the week be a Thursday, and $weekday(T, W), 1 \leq W, W \leq 5$ requires that the time is during the working week (Monday to Friday). Many other calendars and temporal constraints may be defined in a similar way.

In [Bertino et al. 1998], intervals of time like "summer time" and periodic conditions such as "the twentieth of each month (pay day)" are identified as useful for a SA to use to specify an access control policy. These elements have a natural representation in our approach. For instance, $month(T, M), 6 \leq M, M \leq 8$ may be used to represent "summer time" (i.e., the summer months are constrained to be June, July and August). To represent "the twentieth of each month", we may use the constraint: $day(T, D), D = 20$.

6.2 User-Role Assignment in $TRBAC^{P,D}_{H2A}$ Programs

The ura rules in a $TRBAC^{P,D}_{H2A}$ program are defined in terms of the predicates used for accessing components of time, and constraints on these components. The basic integer constraints may be expressed in terms of comparison operators from the set $\{<, \leq, >, \geq, =, \neq\}$ and arithmetic operators from the set $\{\times, \div, +, -, mod\}$.

Example 4. Suppose that Jo is assigned to the role r_2 and then to the role r_3 on alternate months in the interval of time from 2001/01/01 until 2001/12/31. On a Monday, Wednesday and Friday on odd numbered months she is assigned to r_2, and on a Tuesday and Thursday on even numbered months she is assigned to r_3. To represent this information, the following ura definitions may be used:

$$\begin{aligned} ura(jo, r_2, T) \leftarrow\ & 20010101 \leq T, T \leq 20011231, month(T, M), M\ mod\ 2 = 1, \\ & weekday(T, W), 1 \leq W, W \leq 5, W \neq 2, W \neq 4. \\ ura(jo, r_3, T) \leftarrow\ & 20010101 \leq T, T \leq 20011231, month(T, M), M\ mod\ 2 = 0, \\ & weekday(T, W), 2 \leq W, W \leq 4, W \neq 3. \end{aligned}$$

The above rules are not range-restricted (for T), but they will only be used for access control when T is fixed. This will be true of all of the other rules in our $TRBAC^{P,D}_{H2A}$ programs. □

6.3 Permission-Role Assignment in $TRBAC^{P,D}_{H2A}$ Programs

The pra rules in a $TRBAC^{P,D}_{H2A}$ program may be expressed in terms of constraints on roles, access privileges, objects, the mapping predicates $\mathcal{M}$, and constraints on times that are defined by using comparison operators.

Example 5. The following rule may be used to specify that, in the interval from 2001/03/01 until 2002/04/01 (but not during the month of December), write access on object o_1 is assigned to the role r_2 on all working days (except Thursday):

$$\begin{aligned} pra(write, o_1, r_2, T) \leftarrow\ & 20010301 \leq T, T \leq 20020401, month(T, M), M \neq 12, \\ & weekday(T, W), 1 \leq W, W \leq 5, W \neq 4. \end{aligned}$$

□

6.4 Denial-Role Assignment in $TRBAC^{P,D}_{H2A}$ Programs

The dra rules, that represent denials in a $TRBAC^{P,D}_{H2A}$ program, are similar in form to the pra rules.

Example 6. The following rule may be used to represent that between 2001/01/01 and 2001/05/01, write access on object o_1 is denied to members of the role r_1 on Saturdays:

$$dra(write, o_1, r_1, T) \leftarrow 20010101 \leq T, T \leq 20010501, weekday(T, W), W = 6.$$

□

6.5 Representing Derivation Rules in $TRBAC^{P,D}_{H2A}$ Programs

In the temporal access control model described in [Bertino et al. 1998], *derivation rules* are included to permit authorizations to be derived from the presence or absence of other authorizations. A wide range of derivation rules may be represented by using constraints in $TRBAC^{P,D}_{H2A}$ programs.

EXAMPLE 7. To represent that Steve and Jane are assigned to the role r_6 for the same period of time between 2001/03/31 and 2001/06/30, a SA may write:

$$ura(steve, r_6, T) \leftarrow 20010331 \leq T, T \leq 20010630.$$
$$ura(jane, r_6, T) \leftarrow 20010331 \leq T, T \leq 20010630.$$

At any point in time, the SA can determine the set of common roles Steve and Jane are assigned to by checking the answers to the goal:

$$? - ura(steve, R1, T), ura(jane, R1, T).$$

This goal requires the full power of the constraint solver to determine the times when Steve and Jane are assigned to a common role; the answer is expressed in terms of constraints on T. $\Box$

In the temporal authorization model that is described in [Bertino et al. 1998], certain sets of derivation rule are not guaranteed to define a unique set of authorizations (hence the need for the *critical sets* algorithm). The problem of an ambiguous set of authorizations is related to the well-known problems of non-stratified negation (e.g., the absence of a unique *stable model* [Gelfond and Lifschitz 1988]). In our approach, these problems may be treated by writing constraints to simulate the negation required to express access control requirements that might otherwise require that non-stratified programs be used.

EXAMPLE 8. Suppose that a SA wanted to represent the following access control requirement: from 2001/01/01 until 2001/12/31, Tom is assigned to the role r_8 on any day from Tuesday through to Friday. Mo can be assigned to the role r_8 only at those instances of time at which Tom is *not* assigned to r_8. To represent Tom and Mo's assignment to r_8, the SA may use the following rules:

$$ura(tom, r_8, T) \leftarrow 20010101 \leq T, T \leq 20011231,$$
$$weekday(T, W), 2 \leq W, W \leq 5.$$
$$ura(mo, r_8, T) \leftarrow 20010101 \leq T, T \leq 20011231, weekday(T, W), 2 > W.$$
$$ura(mo, r_8, T) \leftarrow 20010101 \leq T, T \leq 20011231, weekday(T, W), W > 5.$$

The SA can check that the derivation rule is enforced by checking that there are no answers to the goal

$$? - ura(tom, r_8, T), ura(mo, r_8, T).$$

An alternative representation of the rule defining Mo's assignment to r_8 that does not require the SA to determine manually the negative of a constraint is as follows:

$$ura(tom, r_8, T) \leftarrow 20010101 \leq T, T \leq 20011231, weekday(T, W), 2 \leq W, W \leq 5.$$
$$ura(mo, r_8, T) \leftarrow 20010101 \leq T, T \leq 20011231, not\ ura(tom, r_8, T).$$

Although this program is non-stratified it is easy to prove that it terminates for all queries.[18] □

6.6 Defining Authorized Access in $TRBAC_{H2A}^{P,D}$ Programs

In this section, we consider the specification of authorization triples by using $TRBAC_{H2A}^{P,D}$ programs.

DEFINITION 6.1. *The authorizations rule for the* $TRBAC_{H2A}^{P,D}$ *programs that we consider is:*

$$authorized(U, A, O) \leftarrow date(T), ura(U, R1, T), active(U, R1), senior\_to(R1, R2), pra(A, O, R2, T), not\ denied(R1, A, O, T)$$

where $denied(R1, A, O, T)$ *is defined thus:*

$$denied(R1, A, O, T) \leftarrow senior\_to(R3, R1), dra(A, O, R3, T).$$

The reading of the *authorized* rule in $TRBAC_{H2A}^{P,D}$ is that a user U has A access on an object O at the time T at which U requests this access (the current time extracted from the system clock by using the *date* predicate) if: U is assigned to a role $R1$ at T; U is currently active in the role $R1$; $R1$ can inherit the A access privilege on O from a role $R2$ in a $TRBAC_{H2A}^{P,D}$ role hierarchy defined by *senior_to*; $R2$ has A access on O at T; and $R1$ is not denied A access on O at T.[19]

It follows from the previous discussion that a $TRBAC_{H2A}^{P,D}$ program Π (as described above) may be defined in the following way.

DEFINITION 6.2. *Let* $\mathcal{UAC}_{H2A}^{P,D}$ *be the finite set of ura clauses in* $TRBAC_{H2A}^{P,D}(\Pi)$*, let* $\mathcal{PAC}_{H2A}^{P,D}$ *be the finite set of pra clauses in* $TRBAC_{H2A}^{P,D}(\Pi)$*, let* $\mathcal{DAC}_{H2A}^{P,D}$ *be the finite set of dra clauses in* $TRBAC_{H2A}^{P,D}(\Pi)$*, let* $\mathcal{DSC}$ *be the finite set of ds facts in* $TRBAC_{H2A}^{P,D}(\Pi)$*, let* $\mathcal{STC}$ *be the set of clauses defining senior_to, let* $\mathcal{AC}$ *be the finite set of active rules and activate facts in* $TRBAC_{H2A}^{P,D}(\Pi)$*, let* $\mathcal{M}$ *denote the definitions of the "mapping" predicates, let* $\mathcal{D}$ *denote the date predicate, let* $\mathcal{AUTH}$ *be the singleton that includes the clause defining authorized, and let* $\mathcal{DENIED}$ *be the singleton set that includes the clause defining denied. Then,* $TRBAC_{H2A}^{P,D}(\Pi)$ *is defined in terms of the 10-tuple:*

$$(\mathcal{UAC}_{H2A}^{P,D}, \mathcal{PAC}_{H2A}^{P,D}, \mathcal{DAC}_{H2A}^{P,D}, \mathcal{DSC}, \mathcal{STC}, \mathcal{AC}, \mathcal{M}, \mathcal{D}, \mathcal{AUTH}, \mathcal{DENIED}).$$

Although we use negation in our definition of *authorized*, $TRBAC_{H2A}^{P,D}$ programs are negation-as-failure safe for queries that fix O. The T argument will always be fixed by *date*, U will be fixed by extracting details of the user identifier from the session or login details, and A will be bound by a constant that denotes the type of access a user requests. The negative *denied* literal is therefore guaranteed to be ground when it is selected for evaluation if O is variable-free. Hence, the computational problem of *unsafe* evaluations [Lloyd 1987] will not arise with $TRBAC_{H2A}^{P,D}$ programs if O is ground. In order to ensure that SAs specify negation-as-failure safe

[18]As such, it is guaranteed to have 2-valued consequences.
[19]In TRBAC policies the conditions included in *active* rules will typically consist of temporal constraints. Hence, temporal constraints on session management may be naturally implemented.

programs, we suggest that *groundness analysis* is performed on a $TRBAC^{P,D}_{H2A}$ program, whenever it is modified, to check for the satisfaction of the negation-as-failure safe property [Marriott et al. 1990].

As long as we can ensure that every goal of the form $authorized(u_m, a_l, o_k)$ terminates for a $TRBAC^{P,D}_{H2A}$ program Π then we can be certain that Π defines a unique, complete and consistent set of authorizations.

DEFINITION 6.3. *The set of authorizations defined by a* $TRBAC^{P,D}_{H2A}$ *program* Π *is complete iff for every ground instance of* $authorized(u_m, a_l, o_k)$ *either* $\Pi^\star \models_3 authorized(u_m, a_l, o_k)$ *or* $\Pi^\star \models_3 \neg authorized(u_m, a_l, o_k)$.

DEFINITION 6.4. *The set of authorizations defined by a* $TRBAC^{P,D}_{H2A}$ *program* Π *is consistent iff there is no* (u_m, a_l, o_k) *triple such that* $\Pi^\star \models_3 authorized(u_m, a_l, o_k)$ *and* $\Pi^\star \models_3 \neg authorized(u_m, a_l, o_k)$

PROPOSITION 6.5. *A* $TRBAC^{P,D}_{H2A}$ *program* Π *where all* $authorized(u_m, a_l, o_k)$ *goals terminate defines a unique, complete and consistent set of authorizations.*

PROOF. Each goal $authorized(u_m, a_l, o_k)$ that terminates either succeeds or fails. Using Proposition 2.19 we have that either $\Pi^\star \models_3 authorized(u_m, a_l, o_k)$ or $\Pi^\star \models_3 \neg authorized(u_m, a_l, o_k)$ (but not both). Hence, the authorization set is complete, unique and consistent. □

An assurance that a $TRBAC^{P,D}_{H2A}$ program defines a logically consistent set of authorizations is only one type of consistency that a SA may wish to have guaranteed. Another important type of inconsistency to be avoided is something that we refer to as *authorization inheritance inconsistency.*[20]

DEFINITION 6.6. *A* $TRBAC^{P,D}_{H2A}$ *program* Π *exhibits authorization inheritance inconsistency iff*

$$\begin{aligned}\Pi^\star \models_3 \; & ura(u_i, r_i, t) \;\wedge\; active(u_i, r_i) \;\wedge\; ura(u_j, r_j, t) \;\wedge\; active(u_j, r_j) \;\wedge \\ & senior\_to(r_i, r_j) \;\wedge\; authorized(u_j, a_l, o_k) \;\wedge \neg authorized(u_i, a_l, o_k)\end{aligned}$$

where $u_i \neq u_j$ *and* $r_i \neq r_j$.

PROPOSITION 6.7. *A* $TRBAC^{P,D}_{H2A}$ *program* Π *exhibits no authorization inheritance inconsistency.*

PROOF. Suppose that Π exhibits authorization inheritance inconsistency. Then, for $\Pi^\star \models_3 authorized(u_j, a_l, o_k)$, $\exists u_j \in U$ and $\exists t \in T$ such that $\Pi^\star \models_3 ura(u_j, r_j, t) \wedge active(u_j, r_j) \wedge pra(a_l, o, r_j, t)$ and $\Pi^\star \models_3 \neg\exists r_k senior\_to(r_k, r_j) \wedge dra(a_l, o, r_k, t)$. For $\Pi^\star \models_3 \neg authorized(u_i, a_l, o_k)$ while $\Pi^\star \models_3 ura(u_i, r_i) \wedge active(u_i, r_i)$ we must have $\exists r_l \in R$ such that $\Pi^\star \models_3 senior\_to(r_l, r_i)$ and $\Pi^\star \models_3 dra(a_l, o, r_l, t)$. As $\Pi^\star \models_3 senior\_to(r_l, r_i)$ and $\Pi^\star \models_3 senior\_to(r_i, r_j)$ it follows that $\Pi^\star \models_3 senior\_to(r_l, r_j)$, but then we contradict the assertion that $\Pi^\star \models_3 \neg\exists r_k senior\_to(r_k, r_j) \wedge dra(a_l, o, r_k, t)$.

□

[20] Informally, a $TRBAC^{P,D}_{H2A}$ program Π exhibits an authorization inheritance inconsistency if members of a role r_j that is junior to a role r_i ($r_i \neq r_j$) have a permission that members of role r_i do not have.

Many other desirable properties of TRBAC programs may be proven in the way that we have described. Proving such properties gives a SA some important assurances about the access control programs that he or she may represent.

6.7 Maintaining an Assignment History in $TRBAC^{P,D}_{H2A}$ Programs

To maintain a history of expired assignments and to define access control requirements in terms of this history, a SA may simply retain *ura*, *pra* or *dra* rules in a $TRBAC^{P,D}_{H2A}$ program. If a history of revocations is to be maintained then a SA may simply modify the definitions of *ura*, *pra* or *dra* (as appropriate) by adding a $T \leq T_{Revoke}$ condition.

7. ACCESS CONTROL BY $TRBAC^{P,D}_{H2A}$ PROGRAMS

When a fixed user u_m makes a request to exercise the a_l access privilege on an object o_k, the goal $authorized(u_m, a_l, o_k)$ is evaluated on a $TRBAC^{P,D}_{H2A}$ program. The example that follows demonstrates what is involved in evaluating access requests. All access requests on $TRBAC^{P,D}_{H2A}$ programs (and other forms of $RBAC$ and $TRBAC$ programs) are evaluated in essentially the same way.

Example 9. Consider the specification of the $TRBAC^{P,D}_{H2A}$ program expressed in the examples from the previous section and suppose that the $RBAC_{H2A}$ role hierarchy that is included in the $TRBAC^{P,D}_{H2A}$ program is defined by the following ds assertion: $ds(r_1, r_2)$. Moreover, suppose that Jo requests write access on the object o_1 on 2001/03/02 (a Friday), and is active in role r_2 at the time of her access request. In this case, the initial computation state is:

$$\langle authorized(jo, write, o_1) \mid true\rangle.$$

After evaluating the *date* subgoal, generated from *authorized*, the computation state becomes:

$$\langle ura(jo, R1, 20010302), active(jo, R1), senior\_to(R1, R2),$$
$$pra(write, o_1, r_2, 20010302), not\ denied(R1, write, o_1, 20010302) \mid true\rangle.$$

Resolving $ura(jo, R1, 20010302)$ with respect to the definition of $ura(jo, r_2, T)$ from Example 4 results in a state:

$$\langle active(jo, R1), senior\_to(R1, R2), pra(write, o_1, r_2, 20010302),$$
$$not\ denied(R1, write, o_1, 20010302) \mid R1 = r_2, T = 20010302, 20010101 \leq T,$$
$$T \leq 20011231, month(T, M), M\ mod\ 2 = 1, weekday(T, W),$$
$$1 \leq W, W \leq 5, W \neq 2, W \neq 4\rangle.$$

This can be simplified by replacing $R1$ by r_2, and removing satisfiable constraints to generate the state:

$$\langle active(jo, r_2), senior\_to(r_2, R2), pra(write, o_1, r_2, 20010302),$$
$$not\ denied(r_2, write, o_1, 20010302) \mid true\rangle.$$

The evaluation of $active(jo, r_2)$ trivially succeeds, the subgoal $senior\_to(r_2, R2)$ succeeds with $R2 = r_2$, and when the definition of $pra(write, o_1, r_2, T)$, from Example 5, is used in the process of evaluating $pra(write, o_1, r_2, 20010302)$ the set of constraints are all trivially satisfiable.

Finally, the negative literal is selected and a sub-computation from the state

$$\langle denied(r_2, write, o_1, 20010302) \,\|\, true\rangle$$

arises. Clearly, we can use negation-as-failure for this negative literal because it has no variables.

In evaluating the sub-computation, there are two solutions found for the goal $senior\_to(R3, r_2)$, where $R3 = r_2$ and $R3 = r_1$. For the first solution, the goal $dra(write, o_1, r_2, 20010302)$ fails because there is no matching definition of dra for r_2. For the second solution, the goal $dra(write, o_1, r_1, 20010302)$ uses the rule from Example 6 to generate the constraints

$$T' = 20010302, 20010101 \leq T', T' \leq 20010501, weekday(T', W'), W' = 6$$

which are simple to solve in the order reached, but fail (because the weekday W' is 5 (Friday)).

As the sub-computation fails, the negative literal succeeds. Overall, the derivation succeeds with the answer *true*. Thus, Jo has write access on o_1 at the time of her access request on 2001/03/02. □

Modern CLP systems optimize the type of computation involved in Example 9 by avoiding using equations where parameter passing is enough, keeping track of fixed variables and removing (or never really adding) trivial constraints to the store (e.g., [Jaffar et al. 1992; Kelly et al. 1998]). This together with indexing on arguments, in order to examine the minimal number of applicable rules when there is a choice, makes the computation very efficient to perform (see Section 10).

Access control queries need to be inexpensive because there may be thousands per second. If a program is negation-as-failure safe then expensive constraint solving algorithms never need to be invoked for access control queries. The remaining expense comes from the search required to determine if an action is authorized, which is (more or less) inherent in the definitions of user-role, permission-role and denial-role assignments, and the seniority relationships in a role hierarchy.

The key results for the operational method used to evaluate access requests follow next. These results are expressed with respect to a $TRBAC^{P,D}_{H2A}$ program that implements denial-role precedence. However, the results that we describe also apply (with minimal changes) to $TRBAC^{P}_{H2A}$ programs, $TRBAC^{D}_{H2A}$ programs, and $TRBAC^{P,D}_{H2A}$ programs with permission-role precedence. The results are applicable when o_k is fixed in $authorized(u_m, a_l, o_k)$.

PROPOSITION 7.1. *(Soundness – successful requests) If $authorized(u_m, a_l, o_k)$ succeeds by the operational method then $\Pi^\star \models_3 authorized(u_m, a_l, o_k)$.*

PROPOSITION 7.2. *(Soundness – failing requests) If $authorized(u_m, a_l, o_k)$ finitely fails by the operational method then $\Pi^\star \models_3 \neg authorized(u_m, a_l, o_k)$.*

PROPOSITION 7.3. *(Completeness – successful requests) If $\Pi^\star \models_3 authorized(u_m, a_l, o_k)$ then $authorized(u_m, a_l, o_k)$ succeeds by the operational method.*

PROPOSITION 7.4. *(Completeness – failing requests) If $\Pi^\star \models_3 \neg authorized(u_m, a_l, o_k)$ then $authorized(u_m, a_l, o_k)$ will finitely fail by the operational method.*

PROPOSITION 7.5. *(Flounder-Free Property) The operational method is guaranteed not to flounder for evaluating any* $authorized(u_m, a_l, o_k)$ *goal on a* $TRBAC^{P,D}_{H2A}$ *program.*

PROOF. The result follows from the fact that $TRBAC^{P,D}_{H2A}$ programs are negation-as-failure safe for fixed u_m, a_l and o_k, and the flounder-free criterion in [Lloyd 1987]. □

The results above express that the operational approach is sound and complete with respect to the logical reading of the program for all goals, in particular the *authorized* goals. The importance of these results in the context of access control are expressed by the following definitions and results.

DEFINITION 7.6. *(Safety Property) If* $\Pi^\star \not\models_3 authorized(u_m, a_l, o_k)$ *(i.e., a user* u_m *is not specified in a* $TRBAC^{P,D}_{H2A}$ *program* Π *as having* a_l *access on object* o_k*) then the goal* $authorized(u_m, a_l, o_k)$ *will not succeed by using the operational method with* Π.

DEFINITION 7.7. *(Availability Property) If* $\Pi^\star \models_3 authorized(u_m, a_l, o_k)$ *(i.e., a user* u_m *is specified in a* $TRBAC^{P,D}_{H2A}$ *program* Π *as having* a_l *access on object* o_k*) then the goal* $authorized(u_m, a_l, o_k)$ *will succeed by using the operational method with* Π.

PROPOSITION 7.8. *The operational approach guarantees that the safety and availability properties are satisfied with respect to any* $TRBAC^{P,D}_{H2A}$ *program* Π.

PROOF. Follows from the Soundness and Completeness results for the operational approach applied to Π. □

The safety property is important from an access control perspective because it ensures that no unauthorized access request is ever permitted. Availability is important because it implies that the evaluation mechanism is sufficiently strong to ensure that all authorized access requests are provable, and equivalently that no authorized access request is ever denied.

The operational approach should be a decision procedure for evaluating access requests. In this respect, we have the following result.

PROPOSITION 7.9. *(Termination) If* Π *is a recursion-free* $TRBAC^{P,D}_{H2A}$ *program and* $authorized(u_m, a_l, o_k)$ *is a goal then the operational approach will succeed or fail after a finite number of derivation steps.*

For proving termination of recursive $TRBAC^{P,D}_{H2A}$ programs, methods based on program analysis may be used [Genaim and Codish 2001; Speirs et al. 1997].

8. REPRESENTATION AND COMPUTATION WITH $TRBAC^{P,D}_{C3A}$ PROGRAMS

In this section, we consider $TRBAC^{P,D}_{H2A}$ programs that are extended to include separation of duties constraints. We also consider constraint checking on $TRBAC^{P,D}_{C3A}$ programs. In the discussion that follows, we only refer to $TRBAC^{P,D}_{C3A}$ programs; the representation of separation of duties constraints and the procedures that are used to check these constraints are common to $TRBAC^{P}_{C3A}$, $TRBAC^{D}_{C3A}$ and $TRBAC^{P,D}_{C3A}$ programs.

As with $RBAC^{P,D}_{C3A}$ programs, to specify pairs of roles that are statically (dynamically) separated a SA will include $ssd(r_i, r_j)$ and $ssd(r_j,r_i)$ facts ($dsd(r_i, r_j)$ and $dsd(r_j, r_i)$ facts) in a $TRBAC^{P,D}_{C3A}$ program or the definition of sym_ssd (sym_dsd).

The semantics of a violation of the ssd constraint on $TRBAC^{P,D}_{C3A}$ programs may be specified in the following way.

Definition 8.1. *The ssd constraint is inconsistent with respect to a state of a $TRBAC_{C3A}$ program Π iff there exists a user u_m ($u_m \in U$) and an instance of time t such that $\Pi^\star \models_3 ura(u_m, r_i, t)$, $\Pi^\star \models_3 ura(u_m, r_j, t)$, and $\Pi^\star \models_3 ssd(r_i, r_j)$ (where $r_i \in R$, $r_j \in R$ and $r_i \neq r_j$).*

The ssd constraint for $TRBAC^{P,D}_{C3A}$ programs may be defined by the rule:

$$inconsistent\_ssd(U, R1) \leftarrow ura(U, R1, T), ssd(R1, R2), ura(U, R2, T).$$

The $inconsistent\_ssd$ rule expresses that the ssd constraint is violated if a user U is specified as being assigned to a statically separated pair of roles at a time T.

The ssd constraint is checked by a SA whenever a definition δ of $ura(U, R1, T)$ is appended to a local copy of the $TRBAC^{P,D}_{C3A}$ program Π.[21] The SA will then evaluate an $inconsistent\_ssd(U, R1)$ query with respect to $\Delta\Pi$ where $\Delta\Pi = \Pi \cup \delta$.

Example 10. Suppose that a $TRBAC^{P,D}_{C3A}$ program Π includes the fact $ssd(r_1, r_2)$, the definition of $inconsistent\_ssd$ for $TRBAC^{P,D}_{C3A}$ programs, and the following ura rule:

$$ura(bob, r_2, T) \leftarrow 20010101 \leq T, T \leq 20011231.$$

That is, Bob is assigned to the role r_2 on every day in the year 2001.

Now, suppose that the following rule is to be inserted into $TRBAC^{P,D}_{C3A}(\Pi)$ to produce the updated form of $TRBAC^{P,D}_{C3A}(\Pi)$, $TRBAC^{P,D}_{C3A}(\Delta\Pi)$:

$$ura(bob, r_1, T) \leftarrow 20010301 \leq T, T \leq 20010331.$$

This rule specifies that Bob is assigned to the role r_1 on every day in March 2001.

The SA evaluates an $inconsistent\_ssd(bob, r_1)$ query on $TRBAC^{P,D}_{C3A}(\Delta\Pi)$. After resolving $ssd(R1, R2)$ in the definition of $inconsistent\_ssd$ with the fact $ssd(r_1, r_2)$ in $TRBAC^{P,D}_{C3A}(\Delta\Pi)$, the computation state is:

$$\langle ura(bob, r_1, T), ura(bob, r_2, T) \mid true\rangle.$$

Evaluating $ura(bob, r_1, T)$ and then $ura(bob, r_2, T)$, using the definitions of Bob's assignment to r_1 and r_2, produces the following set of constraints:

$$20010101 \leq T, T \leq 20011231, 20010301 \leq T, T \leq 20010331.$$

As this set of constraints is satisfiable, it follows that $inconsistent\_ssd(bob, r_1)$ succeeds. Hence, the constraints on $TRBAC^{P,D}_{C3A}(\Delta\Pi)$ are violated and the insert is rejected (as we would expect). □

Constraint checking on $TRBAC^{P,D}_{C3A}$ programs will often involve uninstantiated variables that a logic program could not correctly treat (as in the previous example).

[21] Only insertions of ura rules can violate the ssd constraint.

Indeed, the temporal RBAC models that are represented by logic programs in [Barker 2001] are restricted to representing $TRBAC_{H2A}$ programs precisely because temporal constraint checking involves uninstantiated variables.

The following results apply to the operational method used to check the separation of duties constraints on $TRBAC^{P,D}_{H2A}$ programs.

PROPOSITION 8.2. *(Violation of ssd) If a rule with head $ura(U, R)$ (or a less general instance of $ura(U, R)$) is inserted into a $TRBAC^{P,D}_{C3A}$ program Π then the ssd constraint is violated in the updated state $\Delta\Pi$ of Π iff the evaluation of the query $? - inconsistent\_ssd(U, R1)$ on $\Delta\Pi$ succeeds by the operational method.*

PROOF. Follows from the soundness and completeness of the operational method in the case of success. □

PROPOSITION 8.3. *(Satisfaction of ssd) If a rule with head $ura(U, R)$ (or a less general instance of $ura(U, R)$) is inserted into a $TRBAC^{P,D}_{C3A}$ program Π then the ssd constraint is satisfied in the updated state, $\Delta\Pi$, of Π iff the evaluation of the query $? - inconsistent\_ssd(U, R)$ on $\Delta\Pi$ is finitely failed.*

PROOF. Follows from the soundness and completeness of the operational method in the case of failure. □

COROLLARY 8.4. *The operational method is sufficiently strong to identify all violations of the ssd constraint.*

PROOF. An immediate consequence of the completeness of the operational method applied to $TRBAC^{P,D}_{C3A}$ programs. □

The semantics of a violation of the *dsd* constraint on $TRBAC^{P,D}_{C3A}$ programs may be specified in the following way.

DEFINITION 8.5. *The dsd constraint is inconsistent with respect to a state of a $TRBAC_{C3A}$ program Π iff there exists some user u_m ($u_m \in U$) and an instance of time t such that, at t, $\Pi^\star \models_3 active(u_m, r_i)$, $\Pi^\star \models_3 active(u_m, r_j)$, and $\Pi^\star \models_3 dsd(r_i, r_j)$ (where $r_i \in R$, $r_j \in R$ and $r_i \neq r_j$).*

The *dsd* constraint on a $TRBAC^{P,D}_{C3A}$ program may be represented by the *dsd* rule from $RBAC^{P,D}_{C3A}$, to wit:

$$active(U, R1) \leftarrow not\ inconsistent\_dsd(U, R1), activate(U, R1).$$
$$inconsistent\_dsd(U, R1) \leftarrow dsd(R1, R2), activate(U, R2).$$

The next result expresses that the operational method together with the definition of the *dsd* constraint ensures that the *dsd* constraint is always satisfied in a $TRBAC^{P,D}_{C3A}$ program.

THEOREM 8.6. *No user can be active in a pair of roles specified as being dynamically separated in a $TRBAC^{P,D}_{C3A}$ program, Π.*

PROOF. Consider an arbitrary user u_m and role r_i. By contradiction, suppose that at the time at which u_m activates r_i, u_m is active in a role r_j that is specified as being dynamically separated from r_i. For u_m to activate r_i, $inconsistent\_dsd(u_m, r_i)$ must finitely fail by the operational method. As r_i and r_j are specified as being dynamically separated in Π, the definitions of *active* and *inconsistent_ssd* form a negation-as-failure safe pair of clauses, and the operational method is sound and complete in this case, it follows that the goal $dsd(r_i, r_j)$ must succeed by the operational method. However, if the goal $inconsistent\_dsd(u_m, r_i)$ finitely fails by the operational method then u_m cannot be currently active in r_j. But that contradicts the assumption that u_m is active in r_j at the time at which u_m activates r_i. □

DEFINITION 8.7. *A $TRBAC_{C3A}$ program is a $TRBAC_{H2A}$ program extended to include the definitions of the 3-place inconsistent_ssd and inconsistent_dsd rules, and sets of ssd and dsd facts.*

The following additional points should be noted:

—Although we have only considered separation of duties constraints, other forms of constraint on extended forms of $TRBAC^{P,D}_{C3A}$ program may be treated in the way that we have described;

—As constraint checking is performed by using the query evaluation methods that are used in a CLP system, no special constraint checking methods and no special constraint specification language is required (cf. [Ahn and Sandhu 2000]);

—As recursion is not required to express the separation of duties constraints, the termination results that apply to $TRBAC^{P,D}_{H2A}$ programs extend to $TRBAC^{P,D}_{C3A}$ programs.

9. ADMINISTRATIVE QUERIES AND $TRBAC^{P,D}_{S4A}$ PROGRAMS

In our discussion of $RBAC^{P,D}_{S4A}$ programs, we described how user-role, permission-role, and denial-role reviews may be performed. In our representation of $TRBAC^{P,D}_{S4A}$ programs, it is possible for SAs to evaluate temporal review queries with respect to a $TRBAC^{P,D}_{S4A}$ program.

DEFINITION 9.1. *A temporal review query on a $TRBAC^{P,D}_{S4A}$ program is a query that is expressed in terms of the temporal forms of the ura, pra or dra predicates.*

A temporal review query enables a SA to not only determine the set of roles a user is assigned to, and the set of permissions or denials that have been assigned to a role, but also the times at which a user is assigned to a role, and the times at which an access privilege on an object is assigned to or denied to a role. Moreover, any number of rules may be included in a $TRBAC^{P,D}_{S4A}$ program to provide SAs with other types of policy review options. For example, the following rule may be used by a SA to extract the set of all authorizations that are defined by a $TRBAC^{P,D}_{S4A}$ program at a time T:

$$total\_authorizations(U, A, O, T) \leftarrow ura(U, R1, T), senior\_to(R1, R2), \\ pra(A, O, R2, T), not\ denied(R1, A, O, T).$$

EXAMPLE 11. Suppose that a SA wishes to know the times at which users may write object o_k given the access control information expressed in the previous examples, amongst the answers that are generated for the $total\_authorizations(U, write, o_1, T)$ query is the following constraint:

$$U = jo, 20010301 \leq T, T \leq 20010401, month(T, M), M\ mod\ 2 = 1, \\ weekday(T, W), 1 \leq W, W \leq 5, W \neq 2, W \neq 4.$$

That is, from 2001/03/01 until 2001/04/01, Jo is permitted to write o_1 on Monday, Wednesday and Friday during the months of January, March, May, July, September, and November. Clearly, this could be simplified to

$$U = jo, 20010301 \leq T, T \leq 20010331, weekday(T, W), 1 \leq W, W \leq 5, W \neq 2, W \neq 4$$

as in this case there is only one possible month (March) in the time limit. □

Definition 9.2. *A $TRBAC_{S4A}$ program Π is a $TRBAC_{C3A}$ program that is extended to include rules for enabling an arbitrary number of temporal review queries to be performed on Π.*

As in the case of constraint checking on $TRBAC^{P,D}_{C3A}$ programs, the evaluation of SA queries on a $TRBAC^{P,D}_{S4A}$ program may involve variables that cannot be instantiated prior to evaluation, and may result in answers that are expressed in terms of constraints (as in the previous example). Hence, constraint logic programs may be used to generate answers to SA queries on $TRBAC^{P,D}_{S4A}$ programs in cases where logic programs would not be able to produce solutions. It should also be noted that all of the technical results that we gave for the operational approach with respect to $TRBAC^{P,D}_{H2A}$ programs naturally extend to $TRBAC^{P,D}_{S4A}$ programs.

10. PERFORMANCE MEASURES

We have conducted various performance tests on $TRBAC^{P,D}_{S4A}$ programs used to protect relational and deductive databases from unauthorized access requests. These programs have included a 53 role $RBAC_{H2A}$ role hierarchy that has been represented by using a set of facts to represent all pairs of roles in the *senior_to* relation (a total of 312 *senior_to* facts). We use a materialized form of *senior_to* in our tests. We use a $TRBAC^{P,D}_{S4A}$ program defining the protection applicable to 530 users and 497 objects. There are 650 *ura* rules, 1092 *dra* rules and 1185 *pra* rules.

The experiments were performed using SICStus Prolog 3.7.1 [SICStus 1999] on a Pentium 3, 1 GHz machine with 2GB RAM running SunOS. We built a simple temporal constraint solver in SICStus Prolog using constraint handling rules and the finite domain integer interval solver. We did not implement the more complex constructive negation, but none of the experiments below require it. All results are the average over ten runs.

The first set of results illustrates the speed of answering access queries. We compare two versions of the program: the first version uses the simple solver, whereas the second has the constraint solving specialized to tests (as for these queries the time is always fixed).

We examined access queries for three policies: permitted access ($TRBAC^{P}_{H2A}$), allowed access ($TRBAC^{D}_{H2A}$), and authorized access ($TRBAC^{P,D}_{H2A}$ with denial-role precedence). The average query times are given in microseconds (μs). For each query we illustrate two kinds of access requests: successful requests and failing requests. The averages are taken over each possible successful pair of user and object for successful queries, and every pair of user and object for failed queries. The results for this set of tests are shown in Table IV.

For the failing queries, the execution is dominated by queries that cannot match without considering temporal constraints, so the solver overhead is limited. For

Query type	Average time (μs) Solver	Specialized
`permitted` successful	419	10
`permitted` failed	21	14
`allowed` successful	141	6
`allowed` failed	22	14
`authorized` successful	461	10
`authorized` failed	21	15

Table IV. Access request evaluation times for different policies.

	partial solver		complete solver	
Query	answers	time	answers	time
`ura(U1,R,T),ura(U2,R,T)`	5658	2.0s	5488	9.0s
`ura(`$u1$`,R,T),ura(`$u2$`,R,T)`	5658	0.52ms	5488	1.8ms
`inconsistent_ssd(U,R)`	58	0.18s	58	0.24s
`inconsistent_ssd(`u`,`r`)`	58	0.31ms	58	1.4ms
`pra(O,A,R,T),dra(O,A,R,T)`	969	0.35s	969	1.3s
`pra(o1,A,`r`,T),dra(o1,A,`r`,T)`	969	0.32ms	969	1.3ms

Table V. Example SA review queries.

successful queries the solver overhead is more substantial. We also tried executing the queries with the original recursive definition of *senior_to*, surprisingly there was effectively no difference in the performance measures.

The second set of results relate to SA review queries. We search for pairs of users that can take on the same role at the same time by using the query `ura(U1,R,T), ura(U2,R,T)`, and we give the average results for each checking if a fixed pair of users $u1$ and $u2$ can have the same role at the same time. We use the query `inconsistent_ssd(U,R)`, which searches for all users and roles that are inconsistent with the static separation of duties constraint, and we give the average results for the query `inconsistent_ssd(`u`,`r`)` for each user u and role r. We use the conjoined query `pra(O,A,R,T),dra(O,A,R,T)` to find all objects, access privileges and roles that are both permitted and denied at the same time, and we give the average results for checking for a fixed pair of object and role if they have an access privilege that is both permitted and denied at the same time.

We give the results using two versions of the solver. The partial solver, returns a superset of answers, but never omits answers. The resulting partial answers can then be handed to the complete solver to determine whether they are indeed real.

The results are shown in Table V. Clearly the cost of the complete solver is substantial compared to the partial solver, and the partial solver does not return many "false" positives (only for the `ura` queries). The complete solver is efficient enough for administrator queries because these queries are not executed frequently.

Although our performance analysis suggests that using the temporal constraint solver with $TRBAC_{S4A}^{P,D}$ programs is computationally viable, it is important to note that the costs involved in computing with $TRBAC_{S4A}^{P,D}$ programs will be strongly dependent on the constraints that are used in processing queries on the program. A more mature constraint solver, based on our temporal constraint solver, may be developed, and should lead to substantial further reductions in the cost of query evaluation with respect to $TRBAC^{P,D}_{S4A}$ programs.

11. CONCLUSIONS AND FURTHER WORK

We have shown how a range of RBAC and temporal RBAC policies, that are based on extended forms of the "standard" RBAC models from [Sandhu et al. 2000], our ESFK models, may be represented by using a constraint logic programming approach. The evaluation of user access requests, constraint checks and SA queries may be efficiently solved by using a CLP system, and CLP makes it possible to perform certain computations that could not be performed if logic programs were used for implementation. Access control policies can be flexibly represented by using a small number of rules, and increasingly expressive forms of policy can be represented by incrementally adding rules to the set of rules that is required to represent $RBAC^{P}_{H2A}$ programs. Our representation of access control policies also enables a SA to ensure that implementations of a policy satisfy certain essential properties (e.g., that the policy does not permit any unauthorized access).

In future work we want to consider how other forms of limitation on access (e.g., restricting a user to accessing objects only at certain locations, from certain machines etc.) may be accommodated in our approach.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their suggestions for improving this paper. The authors also thank Arnie Rosenthal and Piero Bonatti for their comments on earlier versions of this paper, and we thank Elisa Bertino for highlighting some key revisions required to a previous draft.

REFERENCES

Ahn, G. and Sandhu, R. 2000. Role-based authorization constraints specification. *ACM Trans. on Information and System Security 3,* 4, 206–227.

Atluri, V. and Huang, W. 1996. An authorization model for workflows. In *ESORICS'96.* LNCS, vol. 1146. Springer-Verlag, 44–64.

Barker, S. 2000a. Data protection by logic programming. In *Proc. 1st International Conference on Computational Logic.* LNAI, vol. 1861. Springer-Verlag, 1300–1314.

Barker, S. 2000b. Protecting deductive databases from unauthorized retrievals. In *DBSec 2000.* Kluwer, 301–311.

Barker, S. 2001. TRBAC^{N}: A temporal authorization model. In *Information Assurance in Computer Networks. Methods, Models and Architectures for Network Security,* H. Gorodetski, V. Skormin, and L. Popyack, Eds. LNCS, vol. 2052. Springer-Verlag, 178–188.

Barker, S. and Rosenthal, A. 2001. Flexible security policies in SQL. In *DBSec 2001.* Kluwer, 167–180.

Bertino, E., Bettini, C., Ferrari, E., and Samarati, P. 1998. An access control model supporting periodicity constraints and temporal reasoning. *ACM TODS 23,* 3, 231–285.

Bertino, E., Bonatti, P., and Ferrari, E. 2000. TRBAC: A temporal role-based access control model. In *Proc. 5th ACM Workshop on Role-Based Access Control.* 21–30.

Bertino, E., Catania, B., Ferrari, E., and Perlasca, P. 2002. A system to specify and manage multipolicy access control models. In *Proc. POLICY 2002.* IEEE Computer Society, 116–127.

Castano, S., Fugini, M., Martella, G., and Samarati, P. 1995. *Database Security.* Addison-Wesley.

Cavedon, L. and Lloyd, J. 1989. A completeness theorem for SLDNF resolution. *JLP 7,* 3, 177–192.

Date, C. 2000. *An Introduction to Database Systems.* Addison-Wesley.

Ferraiolo, D., Cugini, J., and Kuhn, R. 1995. Role-based access control (RBAC): Features and motivations. In *Proc. of the 11th Annual Computer Security Applications Conf.* 241–248.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. 5th International Conf. and Symp. on Logic Programming*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

Genaim, S. and Codish, M. 2001. Inferring termination conditions for logic programs using backwards analysis. In *Proc. 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer, 685–694.

Greco, S., Leone, N., and Rullo, P. 1992. Complex: An object-oriented logic programming system. *IEEE TKDE 4,* 4, 72–87.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems 14,* 3, 339–395.

Jajodia, S., Samarati, P., Sapino, M., and Subrahmaninan, V. 2001. Flexible support for multiple access control policies. *ACM TODS 26,* 2, 214–260.

Kelly, A., Macdonald, A., Marriott, K., and Stuckey, P. 1998. Optimizing compilation for CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems 20,* 6, 1223–1250.

Kowalski, R. 1979. *Logic for Problem Solving.* Elsevier-North Holland.

Lloyd, J. 1987. *Foundations of Logic Programming.* Springer-Verlag.

Marriott, K., Sondergaard, H., and Dart, P. 1990. A characterization of non-floundering logic programs. In *Proc. of the 1990 North American Conference on Logic Programming.* MIT Press.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: an Introduction.* MIT Press.

Przymusinski, T. 1988. On the declarative semantics of deductive databases and logic programming. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan-Kaufmann, 193–216.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence 13*, 81–132.

Sandhu, R., Bhamidipati, V., Coyne, E., Ganta, S., and Youman, C. 1997. The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In *Proc. 2nd ACM Workshop on Role-Based Access Control.* 41–49.

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

Sandhu, R., Ferraiolo, D., and Kuhn, R. 2000. The NIST model for role-based access control: Towards a unified standard. In *Proc. 4th ACM Workshop on Role-Based Access Control.* 47–61.

SICStus 1999. Sicstus prolog home page. http://www.sics.se/sicstus/.

Speirs, C., Somogyi, Z., and Søndergaard, H. 1997. Termination analysis for mercury. In *Proc. 4th International Symposium on Static Analysis*, P. V. Hentenryck, Ed. Lecture Notes in Computer Science, vol. 1302. Springer, 160–171.

Sterling, L. and Shapiro, E. 1994. *The Art of PROLOG.* MIT Press.

Stuckey, P. 1995. Negation and constraint logic programming. *Information and Computation 118,* 1, 12–33.

Woo, T. and Lam, S. 1993. Authorizations in distributed systems: A new approach. *Journal of Computer Security 2,* 2/3, 107–136.