Practical Safety in Flexible Access Control Models

Trent Jaeger
IBM T. J. Watson Research Center
and
Jonathon E. Tidswell
University of New South Wales

Assurance that an access control configuration will not result in the leakage of a right to an unauthorized principal, called *safety*, is fundamental to ensuring that the most basic of access control policies can be enforced. It has been proven that the safety of an access control configuration cannot be decided for a general access control model, such as Lampson's access matrix, so safety is achieved either through the use of limited access control models or the verification of safety via constraints. Currently, almost all safety critical systems use limited access control models, such as Bell-LaPadula or Domain and Type Enforcement, because constraint expression languages are far too complex for typical administrators to use properly. However, researchers have identified that most constraints belong to one of a few basic types, so our goal is to develop a constraint expression model in which these constraints can be expressed in a straightforward way and extensions can be made to add other constraints, if desired. Our approach to expressing constraints has the following properties: (1) an access control policy is expressed using a graphical model in which the nodes represent sets (e.g., of subjects, objects, etc.) and the edges represent binary relationships on those sets and (2) constraints are expressed using a few, simple set operators on graph nodes. The basic graphical model is very simple, and we extend this model only as necessary to satisfy the identified constraint types. Since the basic graphical model is also general, further extension to support other constraints is possible, but such extensions should be made with caution as each increases the complexity of the model. Our hope is that by keeping the complexity of constraint expression in check, flexible access control models, such as role-based access control, may also be used for expressing access control policy for safety-critical systems.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*software configuration management*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

General Terms: Design, Management, Security

Additional Key Words and Phrases: access control models, authorization mechanisms, role-based access control

---

Address: Trent Jaeger, 30 Sawmill River Road, Hawthorne, NY, USA 10532 , Email: jaegert@watson.ibm.com
Address: Jonathon E. Tidswell, Department of Computer Science and Engineering, 2052 NSW Australia, Email: jont@cse.unsw.edu.au. This work was done while the author was on an internship at the IBM T.J. Watson Research Center.

1. INTRODUCTION

An important feature of an access control model is the ability to verify the *safety* of its configurations (i.e., the policies expressed using the access control model). A configuration is said to be *safe* if no rights can be leaked to an unauthorized principal [16]. The verification that a configuration is safe is necessary to ensure that a mandatory access control (MAC) policy, such as multilevel security or separation of duty, is being enforced by the configuration.

Unfortunately, the problem of verifying safety for an arbitrary configuration of a general access control model (e.g., Lampson's protection matrix [22]) was shown to be undecideable [16]. To overcome this problem two approaches have been taken: (1) restrict the access control model, such that safety can be proven in general for that model, or (2) augment the access control model with expressions, typically called *constraints*, that describe the safety requirements of any configuration, such that the safety of each configuration can be verified (i.e., ensure that no right is leaked to an unauthorized principal). The first approach results in specialized models designed for a limited or static policies [6; 10] or models that are difficult to use because it is hard to ensure that the restrictions are satisfied [4]. The second approach suffers from the fact that constraint expression is a difficult task. Because the entities that safety constraints govern are not known *a priori*, we must use a first-order predicate logic rather than propositional logic to express constraints in general. A few logical constraint expression languages have been proposed [2; 7], but such languages are too complex for administrators to determine whether a set of constraints really expresses the desired safety requirements properly. Also, we must be careful in the design of higher-level expression models because approaches may be chosen that are too limited, lack necessary extensibility, and prevent administrators from understanding the relationships between constraints.

The lack of a simple, comprehensive approach to constraints means that restricted access control models are used in safety-critical systems. For example, Bell-LaPadula [6] and Domain and Type Enforcement (DTE) [10] require completely trusted principals to assign subjects and objects to types (or labels). In general, the access control policies are expressed only once by a trusted principal and fixed for the life of the system, so access control policies are safe by definition. However, any flexibility that may be added to these models introduces the possibility of safety problems. For example, the SeaView model [24] generalizes the Bell-LaPadula model to define the extent to which principals of one label may make changes to the assignment of objects and subjects to labels. Therefore, principals are created that can modify the model that are not fully trusted. There are two possible interpretations: (1) that the principals are fully trusted within their domain of administration (the SeaView policy) or (2) that the safety of configuration changes must be verified. Once complete trust is not practical, then there is the possibility of a safety violation, so in practice safety is an issue in many systems.

More flexible access control modeling is often necessary in commercial systems or when more complex safety policies, such as Chinese Wall [11] and Dynamic Separation of Duty (SOD) [36], are required. Such policies require that a principal's or role's rights change dynamically to prevent an unauthorized action (e.g., signing a check). Therefore, configuration changes are part of the application domain. In

addition, the notion that an administrator may not be fully trusted is built into some of these models, such as role-based access control (RBAC) (e.g., ARBAC [33]). To enable the enforcement of safety under these conditions, these models include the concept of constraints whereby the administrators can express explicit tests of whether the current configuration meets the system's safety policy.

We observe that there is a continuum in the trade-off between the expressive power of an access control model and the ease of safety enforcement. In a restricted model, such as Bell-LaPadula, constraints are implicit in the model's definition (e.g., a subject of one label cannot write to any object of a 'lower' security label). Therefore, safety enforcement is trivial, but policy expression is limited. On the other hand, general policy expression models, such as RBAC, make constraints explicit concepts and permit the definition of arbitrary constraints. In this case, the expression of safety requirements has proven to be difficult. However, we have recently found that a variety of common safety policies can be enforced by a few constraint types [38]. Further, we have found that these constraints can be expressed as binary relationships in a graphical access control model [39]. Therefore, we want to determine the extent to which safety verification may be simplified by a graphical access control model whose safety expression is based on these constraint types.

Toward this end, we propose a graphical access control model in which constraints are defined using a small number of relationship types. Constraint expression in this model is simplified in four ways: (1) by defining all constraints in terms of binary relations; (2) by splitting constraint expression into three distinct steps, set identification, input selection, and input comparison; (3) by using the graphical model to do set identification; and (4) by using a small number of set-based operations for constraint input comparison. This contrasts with a rule-based approach in which all these facets of a constraint are lumped into a single rule. We find that the expressive power of our constraint expression model is comparable to that of logical languages, except that we require only a small set of binary relationship types to express the examples.

Using this model, we demonstrate the expression of a variety of constraints collected from the literature [21; 26; 36]. Given these constraint expressions, we can empirically evaluate whether and how a graphical access control model simplifies safety enforcement. We find that all our example constraints can be expressed as binary relationships, except for precondition constraints for which two such relationships are needed. Also, we compare the expression of a constraint in RSL99 to the same constraint using the graphical model [1]. The constraints are the same except that: (1) the specific sets involved in the constraints are indicated graphically by our approach rather than being part of the rule and (2) we define a higher-level comparator function than RSL99 which reduces the length of the statement. However, some constraints require iteration over the members of one set or the other, and the addition of this expression starts to make the constraints complex. Thus, we find that the graphical model enables the expression of a variety of constraints, expression complexity is reduced (but, not always as simple as we would like) because of the graphical expression of some constraint concepts, and the same model may be used for system administrator tasks, internal representation, safety verification, and safety policy analysis. We believe that graphical access control models

can form the basis of an access control framework for enforcing practical safety in general access control models.

The paper is structured as follows. In Section 2, we collect the various safety policies that others have identified in the literature. The properties of these safety policies drive the design of the access control model. In Section 3, we present background on safety enforcement and discuss the effectiveness of previous approaches. In Section 4, we describe our approach to safety verification. In Section 5, we develop our access control model. We begin with a simple model consisting only of the most basic access control concepts. We then define constraints and incrementally add complexity both in terms of constraints, such as universal quantification, and access control representation, such as inheritance. In Section 6, we discuss some key issues about the utility of our proposed access control model, such as computation complexity and maintenance complexity. In Section 7, we conclude and outline future work.

2. EXAMPLE SAFETY POLICIES

Our primary goal is to collect a set of common safety policies and define an access control model in which these policies can be enforced. In defining such an access control model, we prefer concepts that make the expression of such policies relatively easy and the computation of safety (i.e., that the policy is being enforced) efficient. Fortunately, a significant body of access control research has focused on policy. Thus, we can simply gather this knowledge.

However, it is more difficult to define criteria that identify policy expression as 'relatively easy.' In this paper, we propose an approach to access control modeling and constraint expression, and express a variety of common examples using the approach. Verification of the efficacy of the approach is done empirically using these examples and formally by comparing the resultant expressive power and usage of the graphical system in Section 6.

One of the important themes that resonates through (and predates) the literature on RBAC is separation-of-duty or conflict-of-interest constraints [15; 21; 23; 26; 32; 29; 33; 34; 36; 41, for example]. For the sake of clarity, we will present a harmonized merge of the taxonomies by Simon & Zurko [36] and the extensions by Nyanchama & Osborn [26].

In the standard RBAC language [34] the harmonized taxonomies of Simon & Zurko and Nyanchama & Osborn are:

User–user conflicts are defined to exist if a pair of users should not be assigned to the same role. In models extended to support groups of users this extends to not assigning the users to the same group (except a logical group containing everybody).

Privilege–privilege conflicts are defined to occur between two privileges (a privilege is a pair right×object) when they should not both be assigned to the same role.

Static user–role conflicts exclude users from ever being assigned to the specified roles. These constraints are intended to be used to capture restrictions imposed by factors (such as qualification or clearances) that are not in the model.

Static separation of duty exists if two particular roles should never be assigned to the same person.

Simple dynamic separation of duty disallows two particular roles from being assigned to the same person due to some dynamic event (e.g., Chinese Wall).

Session-dependent separation of duty disallows a principal from activating two particular roles at the same time (e.g., within the same session).

Object-based separation of duty constrains a user never to act on the same object twice. They can also be specified to constrain the same role from acting on the same object twice.

Operational separation of duty breaks a business task into a series of stages and ensures that no single person can perform all stages. Thus the roles that are entitled to perform each stage may have users in common so long as no user is a member of all the roles entitled to perform each stage of a business task.

Order-dependent history constraints restrict operations on business tasks based on a predefined order in which actions may be taken. These are a variation of assured pipelines [10] and a potential part of well formed transactions [12].

Order-independent history constraints restrict operations on business tasks requiring two distinct actions (such as two distinct signatures) where there is no ordering requirement between the actions. These are a part of well-formed transactions [12].

In addition, Kuhn [21] published an alternative taxonomy identifying two axes on which to classify such constraints. His first axis – time – is synonymous with static versus dynamic constraints, and is subsumed by the taxonomy of Simon & Zurko. Kuhn's second axis – the extent to which roles involved in mutual exclusion relationships share rights with other roles – has been largely ignored by the RBAC community. We would also like to express mutual exclusion where some basic rights may be shared as well.

There are a few ways in which these constraints may be viewed. For example, the user-user conflict may be between two specific users, two sets of users (i.e., no roles may be shared by either set), or one set of conflict users (i.e., no two in the set may share a role). The expression of these constraints may be quite different. For example, a constraint between two specific users can be expressed in propositional logic, whereas a constraint over all users is generally expressed in a predicate logic. Therefore, we must examine a number of reasonable variations of these constraints in order to get a sense of the flexibility and complexity of constraint expression in the model. The specific variations are described as the examples are developed.

3. SAFETY ENFORCEMENT BACKGROUND

For models that are not safe by definition, some mechanism for verifying the safety of a configuration in that model is necessary. Since Harrison, Ruzzo, and Ullman showed that the safety problem was undecidable [16], research has focused on two areas: (1) determining whether safety could be decided for access control models with limited, but practical, expressive power and (2) defining constraint languages to express verifiable safety requirements. While a number of access control models were designed that enabled polynomial time safety verification, these models were

not used in practice. Therefore, recent research has focused on the development of constraint languages, but thus far no sufficiently simple and expressive language has been proposed.

First, in the case of limited access control models, the *take-grant* model has a linear time safety algorithm, but there is still a significant difference in expressive power between take-grant and HRU [9; 37; 31]. Sandhu *et al* eliminates most of this difference in his models (SPM, TAM, ESPM, and non-monotonic ESPM) [4; 3; 30; 31]. They demonstrate that an access control model could be designed for which safety is efficiently decidable (i.e., in polynomial time) given a few restrictions, which were claimed to be reasonable for almost any policy.

Ultimately, despite proven expressive power and safety determination, these access control models have not been adopted in practice. We claim that there are two primary reasons for the lack of acceptance: (1) these models are rather complex to use, both due to the subtlety of the restrictions and the complex relationship between SPM/TAM types and capabilities and (2) it is difficult to both define the safety requirements and write practical algorithms that enforce these requirements. Simply stating an initial configuration is difficult enough, but system administrators must also define the safety criteria and, thus far, few, practical safety algorithms have been implemented.

In the second approach, constraints have been part of most RBAC models of recent years [7; 8; 25; 33; 34], but with a few exceptions highlighted below they have always been specified using rule-based systems. Unfortunately rule-based systems, while highly expressive, are harder to visualize and thus to use; thus far they have been avoided by practitioners. A common claim is that such rule-based approaches underlie a higher-level expression, but currently there is still no useful approach to either expressing or managing managing constraints.

Ahn and Sandhu [1] propose a limited logical language called RSL99 for expressing separation of duty constraints in a RBAC model (the updated and current version of this language is called RCL 2000 [2]). RSL99 still provides significant expressive power, but remains quite complex. The combination of quantification functions and modeling concept functions makes the constraints expressed in the language difficult to visualize. Thus, this approach is an improvement over a completely general logical language, but it is still too complex.

Nyanchama and Osborn [26] define a graphical model[1] for role-role relationships which includes a combined view of role inheritance and separation of duty constraints based on roles. Recently Osborn and Guo [28] extended the model to include constraints involving users. However, neither the basic model nor the extended model distinguish between accidental relationships and explicitly constructed relationships. Thus, these models do not support policies with a historical component. Furthermore (as Nyanchama and Osborn noted), the lack of object typing in RBAC models makes it hard to model workflow constraints.

We proposed the basis for the model presented in this paper in the Dynamically Typed Access Control model [41]. We demonstrated [38; 39] that it is possible to construct graphical representations for most of these constraints in the context of role-based access control. In this paper, we formalize the semantics of the access

[1] Their graphs are directed acyclic graphs with a top and a bottom.

control model and perform in-depth evaluation of how our common constraints can be expressed using the model.

4. AN ALTERNATIVE SAFETY APPROACH

To be able to use constraints to ensure safety, we must find a suitable formalism to express constraints. In general, constraints in an access control environment are set comparisons. We observe two steps in expressing a set comparison: (1) expressing the sets to be compared and (2) expressing the comparison to be made. When a rule-based predicate logic formalism is used to express a constraint, the distinction between these two steps is not clear. Thus, one goal we have is to simplify the identification of the sets to be compared. Also, the comparisons that are possible in a predicate logic framework are arbitrary. Thus, comparison can be arbitrarily complex, so another goal is to reduce the complexity and the number of types of comparisons.

To address both these problems, we propose the use of a graphical model to express constraints. That is, we extend the 'graphical role model' used with significant success in RBAC to enable the expression of constraints directly. The obvious advantage of a graphical model is as an aid to visualize a system's policy. An administrator can see the roles, users, and permissions of interest, and see whether any constraints are relevant to these entities. Also, it is possible to provide specialized views of the graph that show only the associated information or information of particular interest.

More importantly, the use of a graphical access control model helps us separate the steps of set identification and set comparison in constraints. First, since each node in the graph represents a set, each set involved in a constraint can be represented explicitly by a node. This means that constraints can be simply expressed between the relevant nodes, and administrators can see if they are creating a lot of nodes simply to express constraints (i.e., expressing complex constraints).

Second, since constraints are relationships among nodes in the graph, we can keep the configuration simple by preferring the use of binary constraints. A binary constraint is a comparison between a pair of sets. If all the sets involved in a constraint are precomputed, then in most cases all that is necessary are binary constraints. However, as the examples show, some more complex constraints will be necessary. However, these constraints are required only for more complex concepts than set comparisons (e.g., ordering of set elements).

In the remainder of the paper, we define and demonstrate our approach to safety, using our model for the expression of constraints. We first define the basic concepts of the access control model and define the functions for computing the sets upon which constraints will be based. This basic access control model consists of subjects, objects, and roles. Since subjects and objects may be aggregations themselves quite often, we provide an extensible approach to expressing these aggregations, such that access policy and constraints can be specified in terms of these aggregations. We next define the constraint model which is based on the set operators identified above. We then express constraints and identify the extension necessary to cover the complete constraint space.

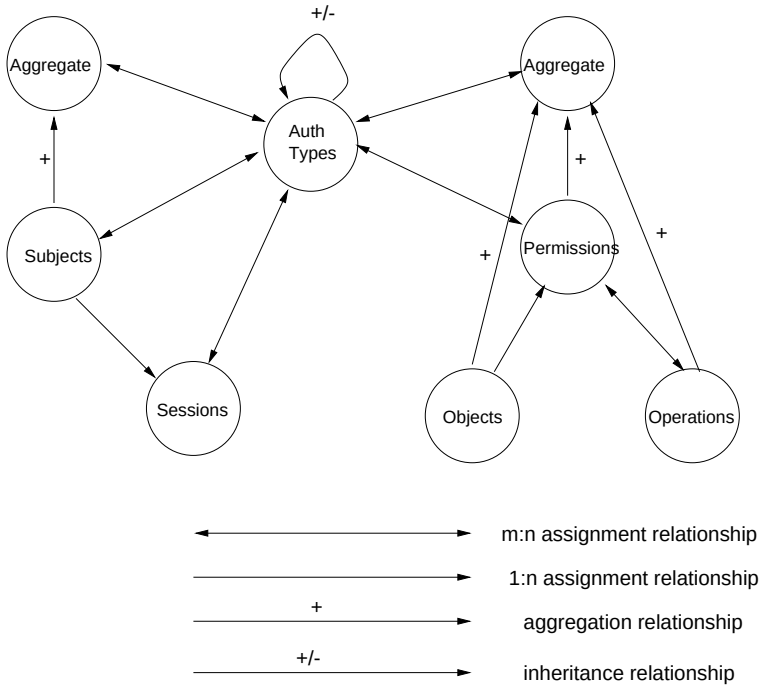

Fig. 1. The node types in the graphical access control model and their assignment relationships.

5. ACCESS CONTROL MODEL

In this section we develop an access control model, including constraints, that supports the expression of the constraints listed in Section 2. We start with a traditional basic access control model consisting of subjects, objects and an authorization relation, and a basic constraint model for this access control model. Both the access control and constraint models are extended to support the expressions necessary to implement the example constraints. These extensions are not ad hoc, however. Rather extensions add new general concepts that were not previously defined in the base model, and we envision that further new concepts may be built from these concepts as well, if necessary.

To give the reader a complete view of the graphical access control model developed, we first define the semantics of the complete model. For those that want to understand the reasoning behind the concepts and semantics, they can read ahead and refer back to the complete definition as necessary.

Definition 1. The graphical access control model is defined by a graph $G = (Y, Z)$ where Y is the set of nodes and Z is the set of edges where each node represents a set and each edge represents a relationship. There are several types of sets and relationships in the model which we define formally. We first define the set types in the model. The sets and the types of assignment relationships between them are shown in Figure 1. The section number indicates the section in which the concept is introduced.

—S is a set of subject types (Section 5.1)

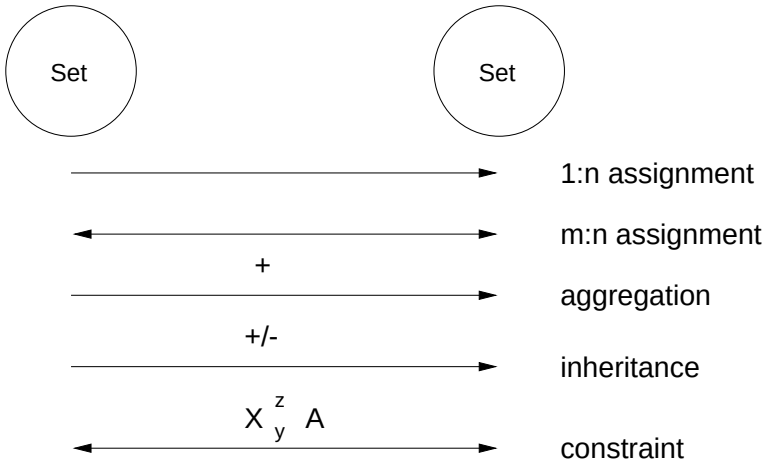

Fig. 2. The relationship types in the graphical access control model.

—P is a set of permission types (Section 5.1)

—T is a set of authorization types (e.g., a role) (Section 5.1)

—X is a set of sessions (Section 5.3)

—O is a set of object types (Section 5.4)

—Op is a set of operations on object types (e.g., rights) (Section 5.4)

—An *aggregate* is a set of elements of the same type (e.g., subjects and objects) (Section 5.5)

—The function $type(y)$ where $y \in Y$ determines the type of the node.

Next, we define the model's relationships and display them in Figure 2.

—A subject assignment SA of subjects to authorization types $SA \subseteq S \times T$. Such an assignment enables the computation of the subjects assigned to a type $S(t)$ and types assigned to a subject $T(s)$. (Section 5.1)

—A permission assignment PA of permissions to authorization types $PA \subseteq P \times T$. Such an assignment enables the computation of the permissions assigned to a type $P(t)$ and the types assigned to a permission $T(p)$. (Section 5.1)

—A subject-session assignment SXA relates a single subject $s \in S$ to a session $x \in X$. This subject is assigned for the lifetime of the session. Such a relation defines the function $S(x) = s$. (Section 5.3)

—A session-type assignment XTA relates a sessions to authorization types $XTA \subseteq X \times T$. Such a relationship enables the computation of the function $T(x)$ which determines the authorization types active in a session. (Section 5.3)

—Variations on the above relationships are also useful, such as the history of assignments (i.e., assignments that have ever existed) and unique applications of assignments (i.e., assignment that have been used in an authorization). For any type in the model *type* and node $z \in Z$, we define functions $type_H(z)$, $type_U(z)$, $type_O(z)$ which compute the history of the assignments of elements of *type* to z, the unique usage of elements of *type* for z, and the order of unique usage of *type*

for z, respectively. Unique usage are the assignments actually used in approved authorizations. Such information is useful for restricting the history and order of authorizations. (Section 5.3)

—An object assignment OA assigns objects to their permissions $OA \subseteq O \times P$. There is a restriction of one object assigned to a permission. Such an assignment enables the computation of the object in a permission $O(p)$ and the permissions to which an object is assigned $P(o)$. (Section 5.4)

—An operation assignment OpA assigns operations to permissions $OpA \subseteq Op \times P$. Such an assignment enables the computation of the operations in a permission $Op(p)$ and the permissions to which an operation is assigned $P(op)$. (Section 5.4)

—Implicit in the model are other assignments, such as permission-subject assignments PSA and object-subject assignments OSA, $PSA \subseteq P \times S$ and $OSA \subseteq O \times S$. These relationships enable the definition of functions $P(s)$ and $O(s)$ which define the permissions and objects accessible to subject s and $S(p)$ and $S(o)$ that determine the subjects that can use the permission p and object o, respectively. (Section 5.4)

—An aggregation relation AR assigns a set to a greater set. For each type Z, we say $AR_Z \subseteq Z \times Z$. Such a relationship enables the computation of $Z(z)$ for any aggregation (i.e., the members of an aggregation of this type). Such a relationship aggregates the elements of the lesser set (i.e., the first Z) into the aggregate (the second Z). (Section 5.5)

—An inheritance relation IR relates authorization types $IR \subseteq T \times T$. Such a relationship aggregates the permissions of the lesser T (i.e., the first) into the superior T, and aggregates the subjects of the superior T into the lesser T. (Section 5.6)

—A binary constraint C is a tuple $C = (z_1, z_2, fi_1(z_1), fi_2(z_2), fs, fc)$ where $z_1, z_2 \in Z$ are nodes in the graph, $fi_j()$ are the identifier functions, fs is the selector function that determines how the set elements are selected for comparison, and fc is the comparator function. The identifier functions return sets given input z_i. The selector function determines elements of the sets are used in the comparison (e.g., iterate). The comparator function takes two sets as inputs and returns a boolean value. A variety of comparator functions are defined for the graphical access control model (see Section 5.2). Note that the form of constraint specification in the model is $X^Z_Y(A)$ where: (1) X is the comparator function $X = \{=, \subset, \subseteq, \supset, \supseteq, \perp, \not\sim, ||\}$ and their negated counterparts; (2) Y represents the identifier function for both identified sets which may be any node type or aggregate $Y = \{S, T, P, O, Op, ...\}$; (3) Z is a facet type of the node type $Z = \{H, U, O\}$; and (4) A is the selection function $A = \{null, < i, i >, < i >, < a, a >, < a >\}$ which indicates set-to-set comparison, element-to-set comparison, set-to-element comparison, element-to-element comparison, aggregate element-to-set comparison, set-to-aggregate element comparison, and aggregate element-to-aggregation element comparison, respectively (see Section 5.5).

—The constraints that are used in safety verification are collected into the set C_s. (Section 5.7)

The model defined above is essentially a role-based access control model extended as necessary to support the types of constraint relationships identified in Section 2. The typical role-based access control model of Sandhu [33] is extended by adding new concepts (e.g., objects) because some example constraints are expressed in terms of objects. Concept functions, such as history, are added because some example constraints are expressed in terms of such functions. Since many other types of ad hoc constraints are possible in general, the effect of constraints on the access control model's complexity can be very significant. A possible conclusion is that constraints are only of limited utility – some other concept may be necessary to limit the complexity of access control models.

5.1 Basic Concepts

The most primitive access control model is the matrix identified by Lampson [22]: $subject \times object \rightarrow rights$. In this model, each cell x in the matrix defines a set of functions: (1) $S(x) = subject$; (2) $O(x) = object$; and (3) $R(x) = \{rights\}$. That is, each cell represents a specific authorization relationship between a subject and object. Note that functions from the subjects' and objects' viewpoints, such as $O(s)$ for $s \in S$ (i.e., the objects to which s has access), are also represented by this model.

Aggregation is a useful concept in access control models, so recent models enable the mapping of sets of subjects to sets of permissions (i.e., objects and the sets of rights available to those objects for the assigned subjects). We use a notation between that of the dynamically typed access control model (DTAC) [41] and the classical RBAC models to express these relationships. First, we define an authorization relation t (i.e., authorization type or role) as a data type with three functions: (1) $S(t) = \{subjects\}$; (2) $P(t) = \{permissions\}$; and (3) $N(t) = name$. In this case, a type represents an authorization relationship between sets of subjects and permissions (i.e., objects and the operations that may be performed on them). Also, the reverse functions for identifying the type assignments of permissions and subjects are also defined, the authorization types of a subject $T(s)$ and the authorization types of a permission $T(p)$.

Note that many access control models are isomorphic to this model at this level of abstraction. For example, roles in role-based access control models are also authorization relations with the same functions. Even multilevel security can be expressed using this model when we view the authorization relationship as a security level.

Visualization of an access control policy is often useful in understanding it. Abstracting the DTAC model defined above, we get a graph as shown in Figure 3, in which elements of the set S are assigned to elements of the set T, and elements of the set P are also assigned to elements of the set T (the assignments are many-to-many).

Note that we are often also interested in the propagation of assignments across the authorization relationship. For any subject $s \in S$, we would want to determine the permissions available to that subject, $P(s) \equiv \bigcup_{t \in T(s)} P(t)$. Similarly, we can determine the subjects to which a particular permission is available. In general, any assignment to an authorization relation can be propagated to concepts on the opposite side of the relation.

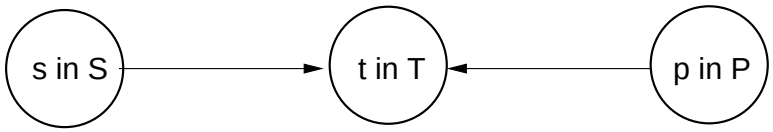

Fig. 3. The basic access control model consists of subjects, permissions, and authorization types (i.e., authorization relationships, such as roles).

5.2 Basic Constraint Model

Since we define our basic model using sets, the natural way to define constraints is as binary relationships between pairs of sets. We chose to limit ourselves to binary relationships for two major reasons: (1) they are easy to describe and draw as labelled edges in a two-dimensional graph, which we hope makes them easier to understand and (2) they are simpler and more compact than ternary (or higher) relationships so the algorithms and data structures are more efficient. In addition, our initial investigations demonstrated that many common constraints can be expressed using only binary relationships [38].

In our model, a constraint consists of functions that identify the sets to be compared and perform the comparison. In Section 5.5, we add another facet to the constraint which defines other ways of using the set elements in comparisons.

The main function of a constraint is to evaluate a comparison. There are two broad categories of constraint comparators. The first is based around the notion of subsets and set equality; thus for example, we have test for *equality* ($=$), *subset* ($\subset$), and *not subset or equal* ($\not\subseteq$). In addition to the standard subset operators we define two sets to be *incomparable* ($\not\sim$) if neither is a subset of the other (except in the degenerate case in which one is empty).

$$A \not\sim B \stackrel{\text{def}}{=} (A \not\subseteq B) \wedge (B \not\subseteq A) \vee (A = \emptyset) \vee (B = \emptyset)$$

The second is based around the notion of overlap between two sets when neither is necessarily a subset of the other, and is defined by limiting *maximal cardinality* of their intersection; so we write $|A \cap B| \leq n$ for two sets A and B. The notion of two sets having no overlap, which we refer to as being *disjoint*, is so common that we give it a special symbol ($\perp$), and write $A \perp B$ for $|A \cap B| = 0$.

It is frequently convenient to denote the application of the same identifier function to both sides of a constraint operator by subscripting the operator with the function name. Thus instead of $P(A) \perp P(B)$ we may write $A \perp_P B$. The most common usage of this is apply to constraints to the objects assigned to a node (subscripted O), the permissions held by a node (subscripted P) or the types assigned to a node (subscripted T).

We note here that conflicts between constraints are possible. For example, one constraint may state that two sets must be equal, but another may state that the same two sets must be disjoint. Conflict resolution strategies for permission assignments have been studied before [8; 20, for example], and in this case defaults, such as *denials take precedence* whereby any negative assignment supersedes a positive assignment, can be used. With constraints, the situation is not so simple, as multiple negative constraints can be violated simultaneously, and no resolution may be

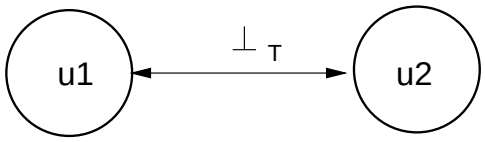

Fig. 4. The graphical representation of Example 1, a *user-user conflict separation of duty* constraint. Subject $u1$ may not be assigned any authorization type to which subject $u2$ is assigned and vice versa. That is, their type sets must have a null intersection.

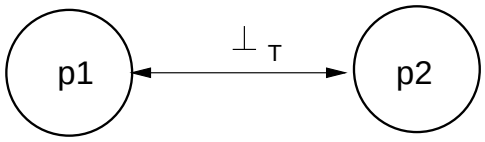

Fig. 5. The graphical representation of Example 2, a *privilege-privilege conflict separation of duty* constraint. Permission $p1$ may not be assigned any authorization type to which permission $p2$ is assigned and vice versa. That is, their type sets must have a null intersection.

possible. In the example above, no resolution is possible: one of the constraints must be modified. Therefore, tools for aiding in conflict detection can be developed, but we do not address this problem in this paper.

We now examine the implementation of the first of our constraint examples.

Example 1. In a user-user conflict separation of duty constraint [36], it is forbidden for two users to both be assigned to any common authorization type. This constraint is enforced by requiring that the authorization type sets of the two users be disjoint as shown in Figure 4, $T(u1) \perp T(u2)$. Also, using our compressed notation we write $u1 \perp_T u2$.

We note at this point that this particular interpretation of this constraint is about as simple as possible. In another interpretation, users from two sets may be restricted from being assigned to any common authorization type. In yet another interpretation, these users may be restricted from being assigned to common authorization types in a set. This constraint is revisited later when we have the tools to express these other variants (Examples 7 and 8).

Example 2. A privilege-privilege conflict separation of duty constraint is similar to Example 1. In this case, we must restrict two permissions from being assigned to a common authorization type. This constraint is shown in Figure 5, $T(p1) \perp T(p2)$ or $p1 \perp_T p2$. Similarly to the user-user conflict constraint, other interpretations of this constraint are possible. Since these interpretations parallel those for the user-user conflict constraint, we do not revisit this constraint.

Example 3. Lastly, we consider a simple separation of duty constraint shown in Figure 6. In this constraint, two roles can never be assigned to the same user. In a simple version of this constraint, we prevent two authorization types that represent the roles from ever being assigned to a common user, $S(t1) \perp S(t2)$ or $t1 \perp_S t2$. Interestingly, we need more expressive power to prevent two types from

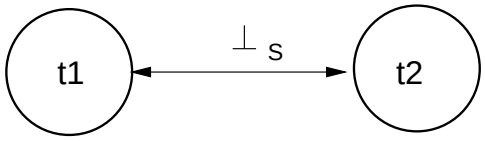

Fig. 6. The graphical representation of Example 3, a *simple separation of duty* constraint. In this constraint, two authorization types are restricted from being assigned to any common subjects. That is, the intersection of their subject sets must be null.

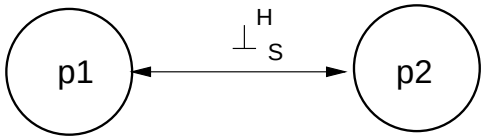

Fig. 7. The graphical representation of Example 4, a *dynamic separation of duty* constraint. In this constraint, a Chinese Wall restriction is to be enforced between the assignment of permissions $p1$ and $p2$. That is, the history of subjects granted permission $p1$ must not overlap with the history of subjects granted permission $p2$.

ever being assigned to one particular user or a particular set of users. We examine this interpretation of the constraint in Example 8.

5.3 Authorization Relationship State

A few security policies (such as Chinese Wall [11] and operational separation-of-duty [36]) depend not only on the current authorization relationship assignments but on other facets of the relationship. We identify the following facets as being useful: (1) the authorizations that are currently activated (*session*); (2) the authorizations that have ever been activated (*history*); (3) the authorizations that have been approved (*unique*); and (4) the list of authorization approvals (*order authorizations*).

The list of currently activated authorization relationships depends on the authorization types that are activated at the present time by the individual subjects. The concept of a *session* has been defined to describe a particular activation of a subjects and a subset of its authorization types. A session $x \in X$ is defined by a single subject $S(x)$ and a set of authorization types activated for that session $T(x)$. Note that the subject assigned to a session cannot be changed. Further, the typical propagation of access control information results in the functions $P(x)$, the permissions available to the session, and $O(x)$, the objects accessible in a session. Effective constraints on sessions require more representation, so we postpone the definition of a session-dependent separation of duty constraint until Section 5.5.

The second type of facet of an authorization relationship is its history. Rather than introducing a new concept for the history of assignments, we introduce new functions which maintain the history of assignments. Such history functions are annotated by a superscripted H, such as $T^H(s)$.

Example 4. We now consider a dynamic separation of duty example (see Figure 7), a simple Chinese Wall policy [11] consisting of a two permissions ($p1$ and $p2 \in$

P) with the restriction that any particular $s \in S$, may access one permission or the other, but not both.

Effectively a Chinese Wall requires there to be no overlap in the subset of S that has ever had active the authorization to access $p1$ with the subset of S that has active the authorization to access $p2$ (and vice versa). Since the history of activated authorizations is a (non-strict) superset of the current active authorizations ($S(A) \subseteq S^H(A)$) we can just compare the historical activations of $p1$ and $p2$, thus we get the simple constraint: $S^H(p1) \perp S^H(p2)$ or $p1 \perp_S^H p2$.

Since permissions are not assigned directly to subjects or vice versa, this is the first constraint whose evaluation is non-trivial. To evaluate the constraint $p1 \perp_S^H p2$ we need to find $S^H(p1)$ and $S^H(p2)$ and verify that the intersection is null. Due to the assignment from S to T it is possible to find $S^H(t)$, for each $t \in T$, and due to the assignment of $p1$ to T it is possible to calculate $T^H(p1)$, $p1 \in P$. For each, $t \in T^H(p1)$ union the values of $S^H(t)$ to get $S^H(p1)$. Perform the analogous calculation for $p2$ to get $S^H(p2)$.

Further, for some constraints we need to track the authorizations requested, perhaps even the order of authorization requests. For example, object-based separation of duty constraints may require knowledge of when an object was used in an authorization (see Section 5.4). Such information should only be maintained if a constraint is defined that requires it. The superscript U identifies an unordered list of authorization approvals, and the superscript O identifies an ordered list of authorization approvals. We define constraints using such concepts in Section 5.7.

5.4 Other Concepts

In addition to constraints on permissions, authorization types, and subjects, constraints may also be expressed about the constituents of permissions: objects and operations (or rights). For example, the object-based separation of duty constraint requires that a user never act upon a particular object twice. Since an object may be accessed by multiple permissions, it is necessary to express this constraint in terms of objects.

Therefore, objects, and likewise operations need to be made first class concepts in the model for constraints to be expressed upon them. We define two new concepts objects O and operations Op, which are composed into permissions, such that every $p \in P$ contains two new functions: $O(p) = o \in O$ and $Op(p) = \{op_1, op_2, ..., op_n\}$ where $op_i \in Op$. That is, a permission consists of one object and one or more operations that can be applied to that object.

Similar to permissions, objects and operations may propagate across the authorization relation, such that we may identify the objects and operations of a subject, $O(s)$ and $Op(s)$, respectively. Note that the operations are dependent on the data type of the object [40]. That is, operations on multiple data types with the same name may exist, so the semantics of the operation depend on the data type and the name of the operation. Therefore, an operation shall consist of both the data type and name.

Example 5. Before we express the object-based separation of duty constraint, we define a constraint analogous to the privilege-privilege conflicts, except it is on objects. For example, we may want to prevent two objects ($o1$ and $o2$) from being

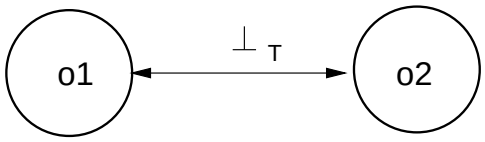

Fig. 8. The graphical representation of Example 5, a conflict constraint between objects. In this constraint, objects $o1$ and $o2$ are restricted from assignment to the same authorization type. That is, there may be no overlap between the authorization types of objects $o1$ and $o2$.

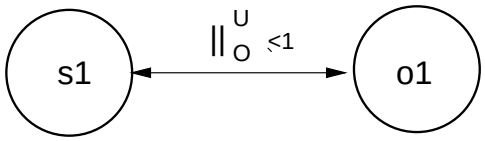

Fig. 9. The graphical representation of Example 6, an *object-based separation of duty* constraint. In this constraint, subject $s1$ is restricted from accessing object $o2$ more than once. Therefore, the cardinality of unique applications of permissions to $o1$ must be no greater than 1.

assigned to a common authorization type. In this case, the constraint is $o1 \perp_T o2$. As shown in Figure 8, this constraint is drawn between the individual objects.

Example 6. In the object-based separation of duty constraint, a subject is restricted from performing an operation on a particular object twice. In Figure 9, we define a constraint between the objects that the subject is restricted from accessing more than once and the subject, $|O^U(s1) \cap O^U(o1)| \leq 1$. This means that the cardinality of the intersection between the subject's unique object accesses and the object accesses on this object is no greater than one. A more compact representation is $s1 \; ||^U_O \leq 1 \; o1$. An edge annotated with this constraint is added to the graph.

5.5 Aggregation and Quantification

Note that as finer-grained constraints are created between concepts, the access control configuration becomes more complex. If we have to express separation of duty at the object level, then many constraints may be necessary. However, aggregation of objects into higher-level concepts enables a reduction in the number of these constraints.

Aggregation applies to any system concept by generalizing each concept slightly to support a value $X(x) = \{x_1, x_2, ..., x_n\}$ where X is the concept type and x and $x_1, x_2, ..., x_n$ are instances of that type. Thus, the concepts which we used before were simply singleton sets. Note that an aggregation relationship in the model will be denoted by a '+'.

The inputs to a constraint comparator are selected from the related sets by a function called the *selection function.* There are three types of selection functions for sets: (1) use the entire set; (2) select each member of the set individually; and (3) select each set that comprised an aggregate. In the first case, the entire set is used in the comparison. For an aggregate, the value selected is the union of each

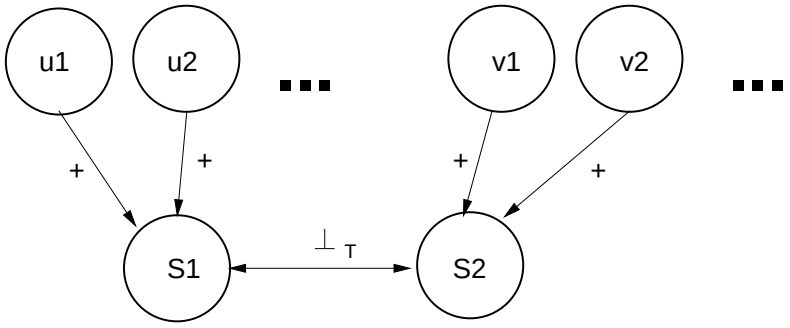

Fig. 10. The graphical representation of an alternative interpretation of the *user-user conflict separation of duty* constraint in Example 7. In this constraint, no subject in $S1$ may assigned to an authorization type to which a member of the $S2$ is assigned. That is, the set of shared authorization types between these groups of subjects must be null.

set that has an aggregate relation with this node. Since we look at each node as a set, we use this as the default semantics, so a special symbol is not defined.

Also, each element of the union may be applied independently in the comparator. Thus, the selection function is an iterator over each element of the set. Since there are two sets in a constraint, we can designate the iteration by $< i, i >$, $< i >$ for iteration over the left-hand set, right-hand set, or both sets in the binary relationship, respectively.

Lastly, we define a selection function that uses the individual sets used to create the aggregate. Thus, the selection function selects each set that was unioned to form the aggregate individually rather than each of the individual members of the resultant aggregation. The comparison is then applied to the members of the set (e.g., authorization types of each subject in an aggregate). Like iteration, these can be indicated for either node in the constraint by $< a, a >, < a >$.

Unfortunately, the semantics of the aggregation relation does not effectively cover all the types of concept grouping in the model. In particular, the functions in authorization types do not behave as summation and iteration functions in all cases. Another relation, called inheritance, is defined for relating authorization types, as discussed in Section 5.6.

We apply aggregation in three ways: (1) to represent sets upon which constraints may be applied; (2) to represent limits on the domain of a constraint; and (3) to represent universal quantification. First, we may state that a constraint is between sets of subjects, objects, permissions, rights, and/or authorization types. Aggregates enable determination of a constraint for all members of a group. Second, a constraint may be applied to a subset of a particular function's values. We use aggregates to define these subsets. Third, universally quantified constraints, such as the requirement that no subject have access to two conflicting roles [1], can be enforced explicitly by associating the set of all subjects with the set conflicting roles, as described below. Thus, an aggregate of all subjects is used to explicitly define universal quantifications.

Example 7. We consider an alternative interpretation of the user-user conflict constraint expressed in Example 1 in which the two sets of users are restricted

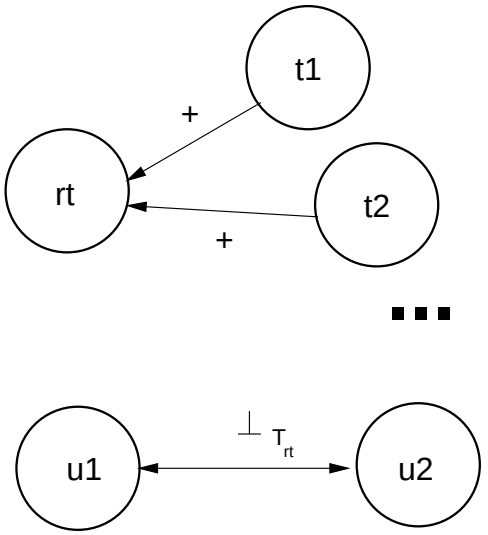

Fig. 11. The graphical representation of yet another alternative interpretation of the *user-user conflict separation of duty* constraint in Example 8. In this constraint, $u1$ and $u2$ must be restricted from sharing an authorization type in the set of restricted types rt. That is, the set of authorization types in rt that are shared between $u1$ and $u2$ must be null.

from being assigned to any common authorization type. This constraint is shown graphically in Figure 10. The users $u1, u2, ...$ and $v1, v2, ...$ are assigned to two separate subject sets $S1$ and $S2$. There is a disjoint relationship on the types that these two sets may be assigned: $S1 \perp_T S2$. In this case, the constraint is between each of the members of the sets $S1$ and $S2$ since $T(S_i)$ is the union of the values of T for each element in S_i.

Example 8. In another interpretation, we want to restrict two users from a particular authorization type from sharing any authorization type from a set of restricted types. In this constraint the restricted authorization types are grouped in an aggregate named restricted types rt. Then, a constraint is made between the two users (or subject aggregates as above), $u1 \perp_{T_{rt}} u2$ as shown in Figure 11. This constraint checks for a null intersection between the types of the two users that are within the restricted types.

In addition, simple separation of duty between two types, $t1$ and $t2$, and a particular set of subjects S_{set} is expressed analogously: $S(t1) \perp_{S_{set}} S(t2)$.

Note that this constraint requires an additional operation to compute. The result of the functions $T(u1) \cap T(u2)$ must be intersected with T_{rt} to determine whether the sets are disjoint with respect to T_{rt}. Fortunately, we can look at this as a new function $T_{rt}(s)$ which defines the subset of T_{rt} that any subject s possesses. Given that $T(s)$ means all the members of the set T that are associated with s, the notion of a function $T_{rt}(s)$ is consistent. With this definition, we are able to preserve the ability to express constraints using a binary relation in Example 8. However, this does increase the complexity of understanding as well as computing the constraint.

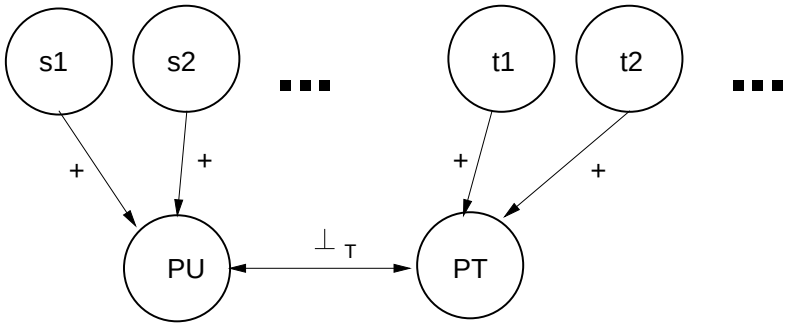

Fig. 12. The graphical representation of Example 9, a *static user-role conflict separation of duty* constraint. In this constraint, a set of users PU are restricted from being assigned an authorization type in set PT. That is, the set of authorization types in PT that are available to any subject in PU must be null.

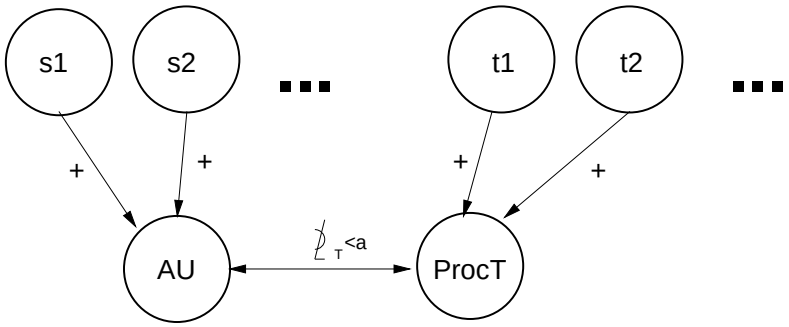

Fig. 13. The graphical representation of Example 10, an *operational separation of duty* constraint. In this constraint, a set of subjects are prohibited from performing all the tasks in a process. In this figure, we assume that each task is assigned to an authorization type in $ProcT$, and each subject's types must not be a superset of $ProcT$.

Example 9. A constraint that we have not demonstrated yet is static user-role conflicts as shown in Figure 12. In this constraint, a user or set of users are prohibited from being assigned to any authorization type in a set. This is easily captured with aggregation by creating aggregates of the prohibited users PU and the set of authorization types to which they are prohibited PT: $T(PU) \perp T(PT)$. This constraint restricts the types of all of the prohibited users from including one of the prohibited roles.

Example 10. Another constraint that we have not demonstrated yet is operational separation of duty. In this constraint, no subject is permitted to obtain all the permissions necessary to perform all the tasks in a process. Typically, each task in a process is represented by an authorization type, so we can express this constraint in terms of these types (called the process types or $ProcT$) for the subject aggregation all users AU: $T(AU) \not\supseteq ProcT$. However, we want to perform this comparison over each subject in the aggregate independently, not on the aggregate

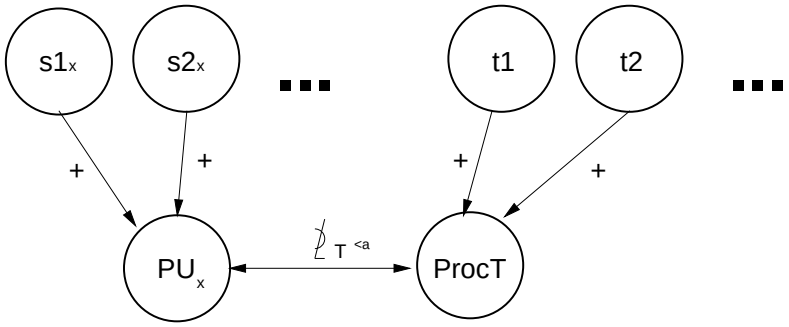

Fig. 14. The graphical representation of a Example 11, a *session-dependent separation of duty* constraint. In this constraint, all the subjects in an aggregate PU are prevented from being assigned to all the authorization types in a restricted set $ProcT$ during their sessions PU_x.

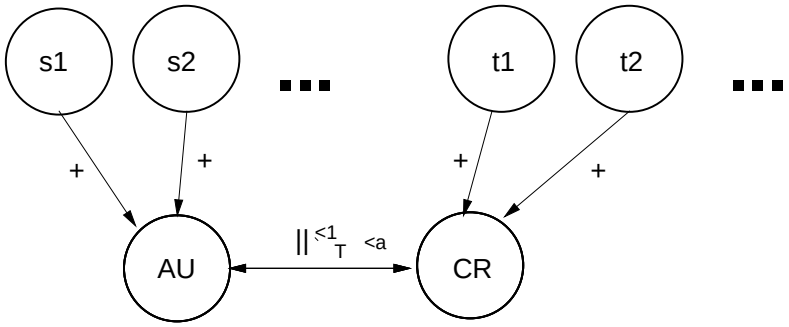

Fig. 15. The graphical representation of Example 12, a universal quantification where all users are restricted from being assigned to more than one conflicting authorization type. In this figure, subjects are automatically assigned to the set AU and we restrict the cardinality of the intersection with the conflicting roles set CR to 1.

itself. The selection function on the constraint is $< a$ since we want to compare the authorization types of each subject aggregated in AU with the set $ProcT$. In Figure 13, the constraint operator, $\not\supseteq_T < a$, indicates the iteration is over each of the aggregated elements of AU.

Example 11. In a session-dependent separation of duty constraint, we want to ensure that no subjects of a particular group are assigned to all the authorization types in a restricted set in the same session. This is the session-based version of Example 10. In order to ease the specification of this constraint, we automatically aggregate the sessions belonging to subjects and their aggregates. Thus, the aggregate of subjects PU may additionally be restricted from access to all types in $ProcT$ in any of their sessions by assigning a subject constraint between their session aggregation PU_x and $ProcT$. This is indicated by a constraint $\not\supseteq_T < a$ which means that no node aggregated in PU_x may have an authorization type set that is a superset of $ProcT$.

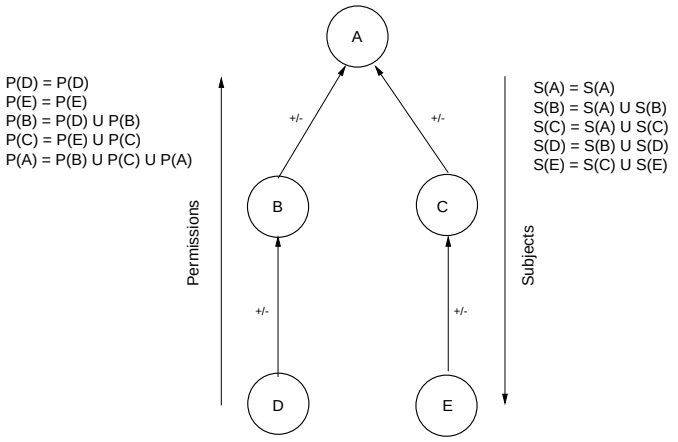

Fig. 16. In an inheritance relation, subjects are aggregated in an inverse direction to permissions.

Example 12. To demonstrate universal quantification, we use a separation of duty constraint demonstrated in the RSL99 constraint language [1] as shown in Figure 15. This constraint states that no user may be assigned to more than one role in a set of conflicting roles. To represent this constraint there are two aggregations: (1) the set of all users (AU) and (2) the set of conflicting roles (CR). Implicitly, each subject can be added to the set of all users, and the system administrators should be able to identify the set of all conflicting roles. Thus, the constraint is between each user in the aggregation and the conflicting roles: $(T(AU) \mid\mid_T \leq 1\ CR)(< a)$. Note that this constraint is an iteration over each member in AU as indicated by the use of the $< a$.

5.6 Inheritance

Unfortunately, the concept of aggregation defined in the previous subsection is not sufficient for authorization types. The problem is that in an authorization type we have two types of assignments, subjects and permissions, that are aggregated inversely to one another. Consider the situation where every subject is an employee, but the permissions of an employee are not the union of the other types. Instead, the expert inherits all permissions, but has the fewest subjects.

We define an inheritance relation, signified by a '+/-', by the direction in which information is transferred by the inheritance relationship as shown in Figure 16. Permission information (e.g., permissions, operations, and objects) is aggregated in the direction of inheritance relationship (the '+'), but subject information is aggregated opposite to the direction of the inheritance relationship (the '-'). Given the direction of the aggregations, the aggregation computation semantics still hold.

Example 13. Given the semantics above, we define a constraint that applies in the context of an inheritance relationship as shown in Figure 17. If we express the constraint in Example 10 in terms of permissions rather than types, then the constraint is: $P(AU) \not\supseteq P(ProcT) < a$. This constraint is basically the same, but we must account for the inheritance of permissions into the types of $ProcT$. The

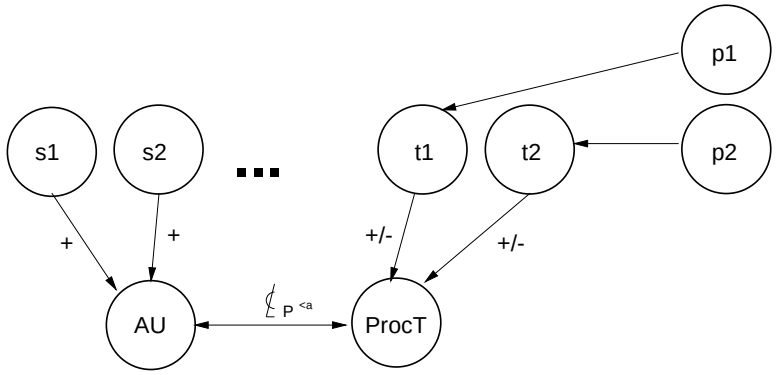

Fig. 17. The graphical representation of another operational separation of duty constraint in Example 13. In this case, we express the constraint in terms of permissions rather than types.

set of permissions are computed according to the inheritance relation.

Note that aggregation and inheritance are both instances of relationships whereby the values of the concept functions (e.g., P, T, and S) in one node are unioned with values of the same concept functions in another node. In the definition of such relationships, the set of concept functions, the direction of combination (e.g., union with the direction of the relationship), and even the combination operator (e.g., union or intersection) may be specified for the relationship. For example, we found that only a subject of the permissions are inherited in MLS policies [39] (e.g., write-up). Thus, a more general expression would include each function type (S, T, P, O, Op), the direction in which the values are modified, and the combination operator. Considering the example separation of duty constraints, we have not yet found the need to generalize our expression language beyond the subject and permission side of the relationship. Nevertheless, such expressions may be necessary and can be added in a straightforward manner.

We now consider the effect of inheritance and aggregation on constraints. First, constraints are aggregated in the direction in which the function is aggregated. If two authorization types are restricted from having a common permission, it is clear that types that inherit permissions from these types must also not have that common permission. Note that due to inheritance the subtypes are also implicitly restricted. Therefore, subject constraints are aggregated in the '-' direction of an inheritance link, and permission constraints are aggregated in the '+' direction. Second, constraints are always relative to the originally related node. As shown in Figure 18, the constraint ρ is relative to B for A and all types aggregating the information relative to the constraint from A.

Example 14. Kuhn [21] identified that there may exist a mutual exclusion between authorization types whereby some permissions not involved in the mutual exclusion may be shared. This constraint is expressed in Figure 19 where a mutual exclusion is set between types A' and B', but the inheritance of this constraint only excludes the permissions of B' from A and A' from B. The constraint is written $P(A') \perp_P P(B')$. The inheritance of the constraint to A is interpreted as

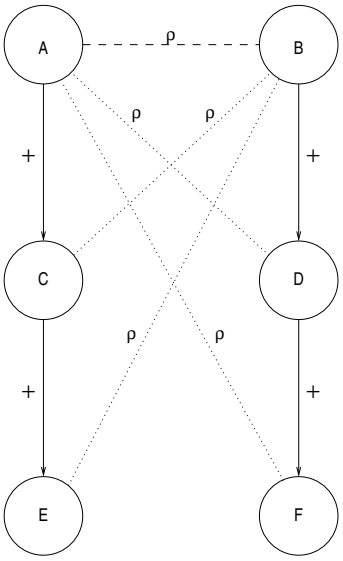

Fig. 18. The constraint ρ is aggregated to its subtypes in the direction of the aggregation of the elements of that function (the '+'). Also, the constraint is relative to the original counterpart node.

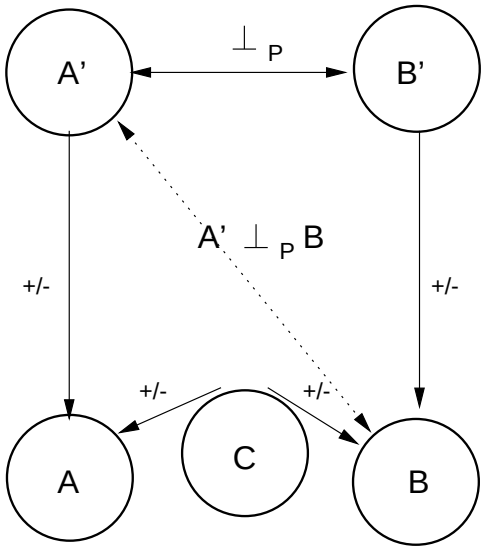

Fig. 19. The graphical representation of Kuhn's *shared rights separation of duty* constraint in Example 14. In this constraint, A' and B' are mutually exclusive on permissions, but A and B are only restricted relative to the permissions assigned to A' and B'. Therefore, they can inherit permissions from C.

$P(A) \perp_P P(B')$. Given this constraint, it is possible for A and B to each inherit permissions from another authorization type C as long as the permissions inherited from C are disjoint from those in A' and B'.

An alternative goal for a constraint is to prevent either A' or its supertypes from subsuming the permissions of B' (e.g., a form of operational separation of duty) or vice versa. An *incomparable* relationship can be used here to prevent either set from subsuming the other. The inheritance of this constraint would prevent the permissions of A and B from subsuming the permissions of B' and A', respectively.

5.7 Order Constraints

The remaining two constraints that we have not examined yet are the history constraints. Both the order-dependent and order-independent history constraints specify that a certain history must have taken place before an operation can be executed. For example, two signature operations must be executed on an object before the approval task can be completed.

In general, these constraints are preconditions. In the order-independent constraints, a set of operations must be executed in any order prior to the execution of constrained operation. In the order-dependent case, the constraint requires that the order of the set of required operations be restricted. Of course, an order can be enforced by specifying the preconditions of each operation to be the set of operations that must have preceded it. Therefore, preconditions based on simple sets are sufficient to implement both order-independent or order-dependent history constraints.

Such history constraints involve individual objects and operations. As such, they depend on the concepts that were introduced in Section 5.4. However, these constraints do not interact significantly with the other separation of duty constraints. The constraints restrict the order of operation execution not the notion of whether the operations can be executed at all. Therefore, extension to history constraints is independent of most of the other extensions.

Example 15. In this example, shown in Figure 20, we describe an order-independent history constraint. In this case, all objects that implement a particular business process $bp1_j \in BP1$ require that the application of operations, $Op^U(bp1_j)$, include a particular set of operations (e.g., two operations of type signature, $sig1$ and $sig2$) before the process completion operation *comp* can be executed. We define a constraint concept $precond(condition, constraint)$ where condition cannot be true unless the precondition constraint is true. In this case, the condition is that the operation *comp* has been executed on any member of $BP1$. The condition is expressed as $Op(BP1) \supset comp < a$. This condition cannot be true until the operations in OP have been executed on the member of $BP1$ first. The precondition constraint is expressed as a relation between the objects in $BP1$ and the operations in OP: $Op_U(BP1) \supset OP < a$ where the OP is the set of operations that must be executed before *comp* (i.e., $sig1$ and $sig2$).

This creates two problems: (1) the first constraint must be named, so it can be included in the constraint and (2) the first constraint only is checked within the context of the precondition, so the context in which it is to be verified must be restricted. The first problem is addressed by permitting constraints to be named.

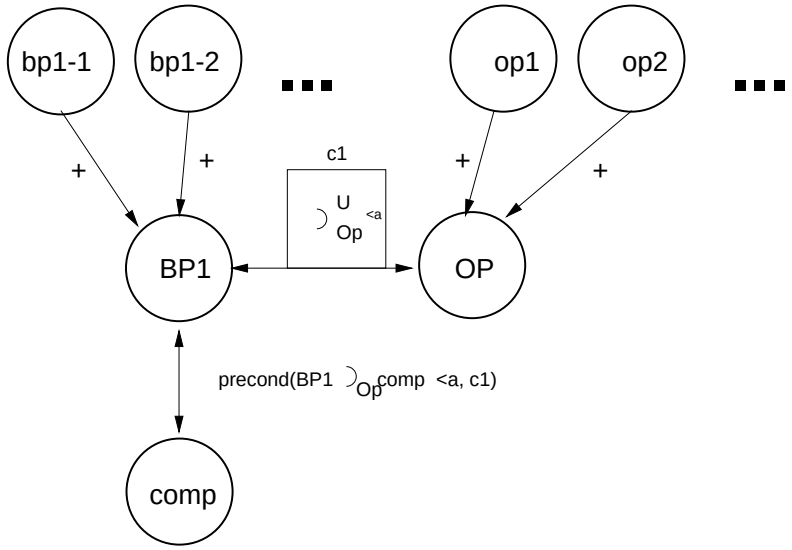

Fig. 20. The graphical representation of an *order-independent history* constraint in Example 15. In this constraint, the operation *comp* cannot be executed on any business process object in $BP1$ until the operations in OP have been executed. Thus, we introduce the notion of a *precondition* constraint that verifies that the unique application of operations on each object in $BP1$ is a superset of OP before *comp* can be executed.

In this case, the first constraint is called $c1$. The second problem is solved by designating which constraints are required for verification in the set C_S. In this case, the precondition must be checked, but $c1$ need not be checked to verify the system's safety.

Clearly, this dependency of a constraint on a constraint complicates the ability to express safety policy. Three new concepts (preconditions, named constraints, and verification constraints) have been added to the model in order to express this constraint properly. Clearly, these constraints are the most complex that we have had to express thus far, so care must be used to avoid a complexity overload.

5.8 Further Possible Extensions

At this point, we have defined all the modeling concepts necessary to express the kinds of constraints that have been identified in the literature. However, we have reason to suspect that other concepts may be useful. Two concepts in particular are parameterized types and triggers. Others include time-based constraints [13].

Besides aggregation and inheritance, another useful approach for grouping concepts is the notion of *parameterized types* [14]. A parameterized type is an authorization type in which facets of the types are defined by parameters. For example, we may define a type $student(course)$ where the rights of the student are defined to be the same except that they are indexed by the course that the student is taking. In this case, students can only access $objects(course)$ for the courses to which they are legitimately registered. We have used parameterized types in collaborative systems [19] and virtual university systems [18].

At this point, we have not studied the extensions necessary to support constraints over parameterized types. Our intuition is that binary constraints for parameterized types can aggregate a great deal of information. However, exceptions to the parameterized types are harder to understand and manage than other aggregations. Thus, parameterized types should probably be used where no or few exceptions are necessary.

The second concept is the notion of triggers. A trigger is the opposite of a precondition. Rather than precluding an operations execution, a trigger is executed upon the execution of an operation. For example, dynamic constraints, such as dynamic separation of duty, may require a change in the set of constraints in a configuration given the execution of an operation. We have resisted the demand to add triggers to the model, as triggers would be similar in complexity to preconditions, which are the most complex concept in the model. Thus far, we feel that triggers are not necessary, but we may be proven incorrect. However, like preconditions, triggers should be used in a limited way to maintain manageability of the model.

6. MODEL EVALUATION

In this section, we evaluate the effectiveness of the graphical access control model for enforcing safety. We begin by evaluating the expressive power of the graphical access control model's constraint specification. We compare the expressive power of the graphical constraint model to that of a constraint language RSL99 [1]. In general, we find that the graphical constraint model makes some simplifications that reduce expressive power, but our empirical and analytical assessments result in the expectation that this additional generality of RSL99 adds little benefit. We then define safety verification using the graphical access control model and evaluate the computational complexity of verifying the example constraints using the model. While no comparable analyses of other systems exist, we believe that the complexity of constraints in this model is typical, and the use of caching can reduce the complexity to near minimal for these constraints (although management complexity would be increased).

6.1 Expressive Power

An interesting issue is the expressive power of this access control model. In the simplest access control model conceivable, constraints are expressed purely in terms of subject sets and permission sets. For example, constraints are expressed by listing the set of permissions that each subject may be assigned (i.e., expression using propositional logic). This model has the nice feature that it is fail-safe in that only permissions that are allowed to be assigned to a subject may be. On the other hand, constraint specification is tedious and dynamic creation of objects requires the creation of new constraints before any rights may be assigned.

On the other hand, a fully general model enables the use of universal and existential quantification on an arbitrary number of variables. Also, the full power of predicate logic would be available.

The graphical constraint model (i.e., the constraint expression subset of the graphical access control model) contains the ability to express universal quantification over two sets, and provides set operations for use on the result. The only predicate in the model is *precond*, and even this has been added with some trepi-

dation.

A question is how close the expressive power of the graphic constraint model is to what is necessary in practice. Unfortunately, this is difficult to prove analytically. As one test, we show empirically in this paper the variety of practical constraints that can be expressed using the model.

Second, we compare the expressive power of the graphical constraint model to another recent proposal for a practical constraint model, RSL 99 [1]. RSL99 supports a restricted first-order predicate logic in which a universal quantification over a predefined set of functions can be specified. The functions in RSL99 correspond to those in the graphical constraint model, except that the different dimensions of the functions, such as history, are not included. Thus, we expect that constraints based on these other dimensions will be more complex to specify. Clearly, RSL99 can express n-ary constraints, whereas the graphical constraint model can only specify binary constraints. However, at this point, we see most practical constraints as comparisons of two concepts: one set has a constraint in relation to another. The addition of ternary and greater concepts makes the language more complex, and we have reservations about system administrators' abilities to express higher-order constraints.

RSL99 also includes typical set operations to create the sets used in the constraint comparison. We define aggregation and inheritance relations to the *union* operator to sets. We did not define an *intersection* operator for the graphical constraint model, as none of the example constraints warranted it. However, it can be added in a straightforward way.

RSL99 uses general, mathematical expressions for performing set comparisons. Therefore, arbitrary set comparisons can be made. In the graphical constraint model, we define a set of higher-level *comparators* that represent common mathematical expressions that are relevant to the example constraint types. For example, the *disjoint* comparator expresses a null intersection. We believe that maintaining a small set of intuitive comparators that cover the range of useful constraints will be key to model simplicity.

RSL99 also includes operators to ease the expression of universal and existential quantification *oneelement* and *allother*, which take one element from a set and repeatedly extract all others, respectively. Our selection functions for sets and elements enable the same information to be expressed. In addition, we have an additional concept which is the iteration over the sets in an aggregation which we found useful in a number of cases.

Ultimately, a comparison of constraint expression between the graphical access control model and RSL99 will require some empirical analysis of how different useful constraints are specified. Example 12 is the expression of a constraint that is also expressed in RSL99 [1]. The RSL99 expression for this constraint is $\mid roles*(OE(U)) \cap OE(CR) \mid \leq 1$. The constraint expression in the graphical access control model is also somewhat complex (see Figure 15). However, the graphical representation reduces some expression complexity by defining the || comparator rather than requiring the full intersection expression in RSL99. Also, the graphical representation of the sets involved in the constraint eliminates the need to express that part of the constraint in the language. Thus, the same expression in RSL99 is much shorter in our model. Quantification is still somewhat complex in both

expressions, and we feel that more work is still necessary to make quantification manageable.

In addition, the graphical model has another potential advantage that we are just beginning to leverage. Unlike a rule, the concepts in a graphical constraint have well defined semantics in the model, so a variety of analysis are possible. For example, we use the graphical access control model to estimate the complexity of safety verification using constraints [17]. Further analyses, such as the identification of redundant or conflicting constraints, may also be useful.

In summary, we think that the graphical constraint model and RSL99 share a great deal of common semantics about expressing access control constraints. The main differences are: (1) that the graphical access control model separates the steps of identifying the sets for comparison, selection the comparison inputs, and performing the comparison and (2) in the trade-off between expressive power and complexity for concepts such as quantification and set comparison and the means of expression of the constraints. In the graphical access control model, these steps in a constraint comparison are more explicit separate. The separate step of set identification is made straightforward in the graphical access control model, in particular. RSL99 provides more flexibility at a cost of additional complexity. We found this additional complexity unnecessary for our example constraints and are striving to keep the number of useful comparators small. We believe that more complex relations are beyond practical application, and the use of general mathematical expressions for set comparison deter from an administrator's ability to intuit the mean of a constraint.

6.2 Safety Verification

Safety verification involves computing all the constraints to determine if the comparisons are satisfied or not. The computation of a constraint involves the three steps made explicit by our constraint definition.

—**Identification**: First, the sets in the constraint are identified. This task involves using the functional definitions of the two sets in the binary relation to compute their membership. Such a computation may be optimized by effective caching of intermediate results.

—**Selection**: Next, the selection function determines how many comparisons will be necessary in order to verify the constraint and what those comparisons will be.

—**Comparison**: Lastly, perform the comparison(s) on the selected inputs. The comparator function is executed on the inputs selected by the selection function.

The computational complexity of safety verification per constraint is the sum of the identification complexity and the product of selection cost and comparison cost.

6.3 Computational Complexity

Any constraint model should enhance the performance of computing constraints. Again examining a trivial constraint model, the worst case computation time to verify a safe configuration is $O(|S||P|^2)$. For each subject, we must determine whether their permissions are in the set of legal permissions.

Example	*Identify*	*Select/Compare*
1-2	$O(1)$	$O(T^2)$
3-4	$O(1)$	$O(S^2)$
5	$O(1)$	$O(T^2)$
6	$O(1)$	$O(O)$
7	$O(ST)$	$O(ST^2)$
8	$O(1)$	$O(T^2)$
9-10	$O(ST+T)$	$O(ST^2)$
11	$O(XT+T)$	$O(XT^2)$
12	$O(ST+T)$	$O(ST^2)$
13	$O(SP+TP)$	$O(SP^2)$
14	$O(P)$	$O(P^2)$
15	$O(O*Op)$	$O(O*Op)$

Table 1. Worst-case computational complexity for each example constraint.

In Table 1, we list the worst case computation time of each of the example constraints. The only constraint that is, in worst-case, as expensive as the simple constraint model, is Example 13, because this is a constraint between subjects and permissions as in the simple model. Constraint 11 is also interesting in that it is computed on the set of sessions as expressed in the worst-case analysis, which is larger than the set of subjects. However, the actual enforcement will be done per session (i.e., one session at a time). When a type is added to a session, it must be verified that the session types is not a superset of the process types ($O(T^2)$).

We briefly examine the computational complexity of the other examples. Examples 1 through 5 are simply intersection computations on a particular pair of sets. Therefore, they are all $O(n^2)$ where n is the size of the particular sets. Example 6 simply examines the objects that have been accessed by the subject to determine if a certain one has already been accessed. Example 7 may require that the unique type assignments must be collected (i.e., unioned) for each user, so its worst-case runtime is dominated by this. Example 8 requires two successive intersections ($T(u1) \cap T(u2) \cap T_{rt}$), but the runtime of each intersection is the same. Examples 9, 10, and 12 may require the collection of types for each user, so their runtime is the same as Example 7. Example 14 may require that we collect all the operations that the user has run, and intersection them with the precondition operation set.

One potential advantage of a graphical model is that we can direct the caching of the identified sets. Note that the selection and comparison complexities always are greater than that of identification, so the benefit of caching is limited to the constant factor of the computation. There are two problems that we must address in caching data: (1) determining what data is a candidate for caching and (2) determining whether caching that data provides a benefit. In the first case, as we see in Example 7, maintaining the set of types of all subjects in an aggregation at the aggregations $S1$ and $S2$ reduces the need to gather this information from each subject (i.e., reducing the selection cost to $O(1)$). Since there is a constraint on types on this aggregation, this indicates that maintaining the value of this function

locally at the aggregation may improve the performance of the constraint check. On the other hand, maintaining the consistency of a set of distributed caches can be expensive as well. Therefore, it does not make sense to cache data that changes much more frequently than the configuration itself, such as activated permissions (e.g., Example 15).

7. CONCLUSIONS AND FUTURE WORK

In this paper, we examine the problem of making safety verification practical for general access control models, such as role-based access control. Previous safety constraint expression approaches appear to be too complex for average system administrators, so we define a graphical access control model that is designed to simplify constraint expression. Constraint expression in this model is simplified in three ways: (1) by splitting constraint expression into three steps, set identification, input selection, and input comparison; (2) using the graphical model to do set identification; and (3) using a small number of set-based operations for constraint input comparison.

We have demonstrated this model on a variety of constraints taken from the literature to show how constraints can be expressed and motivate the selection of concepts for the model. All these constraints can be expressed, but some expressions, particular those requiring iteration over a set, start to become complex. Some further syntactic sugar may be useful here. However, the constraint expression requires much less language expression than even simplified constraint languages like RSL 99.

We also evaluate the expressive power of the model. Our model is restricted along two dimensions: (1) all constraints must be expressed as binary relationships and (2) only a limited number of constraint relationships (i.e., set comparator functions) are available. Neither of these restrictions has prevented us from expressing the constraints, but both are useful, and in the case of binary relationships necessary, in reducing expression complexity.

Lastly, we identify another feature of graphs that may be useful in managing safety policy: the constraint expressions themselves may be reasoned about. This may enable us to automate some management tasks, such as redundancy and conflict detection, and perform some new tasks, such as complexity management. We will pursue the goals in the future.

8. ACKNOWLEDGMENTS

The authors would like to thank John Potter, Leendert van Doorn, Peter Gutmann, and many others who have given us feedback on this work. The authors would particularly like to thank Paul Karger for encouraging us to focus on guaranteeing safety in general access control models.

REFERENCES

[1] G. Ahn and R. Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the* 4^{th} *Workshop on Role-Based Access Control*, 1999.

[2] G. Ahn and R. Sandhu. Role-based authorization constraint specification. *ACM Transactions on Information System Security (TISSEC)*, 3(4), Nov. 2000.

[3] P. Ammann and R. Sandhu. One-representative safety analysis in the non-monotonic transform model. In *Proceedings of the* 7^{th} *IEEE Computer Security Foundations Workshop*, pages 138–149, 1994.

[4] P. E. Ammann and R. S. Sandhu. Safety analysis for the extended schematic protection model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1991.

[5] P. E. Ammann and R. S. Sandhu. The extended Schematic Protection Model. *Journal of Computer Security*, vol. 1, 1992.

[6] D. Bell and L. La Padula. Secure Computer Systems: Mathematical Foundations (Volume 1). Technical Report ESD-TR-73-278, Mitre Corporation, 1973.

[7] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information System Security (TISSEC)*, 1(2), Feb. 1999.

[8] E. Bertino, S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Unified Framework for Enforcing Multiple Access Control Policies. In *Proceedings of ACM SIGMOD Conference on Management of Data*, May 1997.

[9] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the* 7^{th} *ACM Symposium on Operating System Principles*, pages 45–54, 1979.

[10] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the* 8^{th} *National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[11] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 215–228, Oakland, CA, May 1989.

[12] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, April 1987.

[13] A. Gal and V. Atluri. An authorization model for temporal data. In *Proceedings of the* 7^{th} *Conference on Computer and Communication Security*, 2000.

[14] L. Giuri and P. Iglio Role templates for content-based access control. In *Proceedings of the* 2^{nd} *Workshop on Role-Based Access Control*, 1997.

[15] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.

[16] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[17] T. Jaeger. Managing Access Control Complexity Using Metrics. *Proceedings of* 6^{th} *ACM Symposium on Access Control Models and Technologies*, May 2001.

[18] T. Jaeger, T. Michailidis, and R. Rada. Access control in a virtual university. *Proceedings of* 5^{th} *IEEE International Workshop on Enterprise Security (WETICE 1999)*, June 1999.

[19] T. Jaeger, A. Prakash, J. Liedtke, N. Islam. Flexible control of downloaded executable content. *ACM Transactions on Information and System Security (TISSEC)*, 2(2), May 1999.

[20] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[21] D. R. Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in a role-based access control system. In *Proceedings of the* 2^{nd} *ACM Role-Based Access Control Workshop*, 1997.

[22] B. W. Lampson. Protection. In *Proceedings Fifth Princeton Symposium on Information Sciences and Systems*, March 1971. reprinted in Operating Systems Review, 8, 1, January 1974, pages 18 – 24.

[23] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management *IEEE Transactions on Software Engineering*, 25(6), November/December 1999.

[24] T. Lunt, D. Denning, R. Schell, M. Heckman, and W. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[25] E. C. Lupu and M. Sloman. A policy based role object model. In *Proceedings of the* 1^{st} *IEEE Enterprise Distributed Object Computing Workshop*, October 1997.

[26] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security (TISSEC)*, 2(1), Feb 1999.

[27] S. Osborn. Mandatory access control and role-based access control revisited. In *Proceedings of* 2^{nd} *ACM Workshop on Role-Based Access Control*, November 1997.

[28] S. Osborn and Y. Guo. Modelling users in role-based access control. In *Proceedings of the* 5^{th} *ACM Role-Based Access Control Workshop*, July 2000.

[29] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), September 1975.

[30] R. S. Sandhu. The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes. *Journal of the ACM*, 35(2):404–432, 1988.

[31] R. S. Sandhu. The Typed Access Matrix Model. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1992.

[32] R. S. Sandhu. Transaction Control Expressions for Separation of Duties. *Proceeding of the Fourth Aerospace Computer Security Applications Conference*, December 1998.

[33] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information System Security (TISSEC)*, 1(2), Feb. 1999.

[34] R. S. Sandhu, E. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[35] R. S. Sandhu, E. J. Coyne, H. F. Feinstein, and C. E. Youman. Role-based access control: A multi-dimensional view. In *Proceeding of the* 10^{th} *Annual Computer Security Applications Conference*, December 1994.

[36] R. Simon and M. E. Zurko. Mutual exclusion of roles as a means of implementing separation of duty in a role-based access control system. In *Proceeding of the* 10^{th} *IEEE Computer Security Foundations Workshop*, June 1997.

[37] L. Synder. On the synthesis and analysis of protection systems. In *Proceedings of the* 6^{th} *ACM Symposium on Operating System Principles*, pages 141–150, 1977.

[38] J. E. Tidswell and T. Jaeger. Integrated Constraints and Inheritance in DTAC. In *Proceedings of the* 5^{th} *ACM Role-Based Access Control Workshop*, July 2000.

[39] J. E. Tidswell and T. Jaeger. An Access Control Model for Simplifying Constraint Expression. In *Proceedings of the* 7^{th} *ACM Conference on Computer and Communication Security*, November 2000.

[40] J. E. Tidswell, G. Outhred, and J. Potter. Dynamic rights: Safe extensible access control. In *Proceedings of the* 4^{th} *ACM Role-Based Access Control Workshop*, November 1999.

[41] J. E. Tidswell and J. Potter. A dynamically typed access control model. In *Proceedings of the Third Australasian Conference on Information Security and Privacy*, July 1998.