Integrated Constraints and Inheritance in DTAC

Jonathon E. Tidswell and Trent Jaeger
IBM T J Watson Research Centre,
Hawthorne NY 10532 USA,
{jont,jaegert}@us.ibm.com

March 5, 2000

Abstract

Inheritance and constraints are two common techniques for safely managing the complexity of large access control configurations. Inheritance is used to help factor the model, while constraints are used to help ensure that the complexity will not result in an unsafe configuration arising in the future evolution of the system. In this paper we develop an integrated mathematical approach to defining both inheritance and constraints in the dynamically typed access control (DTAC) model. In the process we identify several useful relationships among DTAC objects. The combination of DTAC and these relationships enables us to graphically construct a greater variety and complexity of efficiently verifiable constraints than any other model we are aware of.

1 Access Control

Access control models are basically a way to define a framework to support authorisation decisions of the form

1. Does entity x have right r to entity y ?
2. Who is authorised to give entity x right r to entity y ?
3. Will entity x ever acquire right r to entity y other than by being given it by somebody who is authorised ?
4. What rights does entity x have to which entities ?
5. Which entities have right r to entity y ?

Unfortunately in large access control models these questions become hard to answer and the answers hard to interpret simply due to the scale, and in dynamic systems these questions can be NP-hard [HRU76]. Therefore many models have been developed to structure the authorisation framework so that the questions

can be easier to answer, and the answers easier to interpret. There are two common approaches to structuring the access control model; the first, typically used by capability models, is to limit the number of entities that need to be considered; the second, typically used by ACL based models, is to introduce some structure or grouping mechanism so that questions can be asked about a conceptually smaller and simpler model.

Dynamically typed access control (DTAC) models [TP98] follow the latter approach by introducing types to group collections of entities with similar properties (both security properties and structural properties). By defining abstract types it is also possible to group rights to sets of types of entities, but it then remains to define the relationships between abstract types and "concrete" types. Unfortunately attempting to define relationships opens Pandora's box of problems (see Sections 5 and 8). We present a mathematical approach to defining relationships available in DTAC and relate these to relationships defined in the RBAC model [SCFY96]. We believe that our approach is novel and that as a result we can identify several new relationships with immediate uses.

There are four significant reasons that we began this work:

- to develop both a constraint mechanism and an inheritance model for our DTAC model;
- to explore the parallels we saw between the mathematics underlying our proposed models for both inheritance and constraints;
- to leverage the benefits from a combined approach to develop a constraint model which is simpler than previous RBAC constraint models with an efficient evaluation algorithm that still allowed us to specify complex constraints; and
- to find graphical representations for a larger number of constraints.

We believe we have achieved our goals: our DTAC relationships provide a simple graphical model of access control that includes inheritance and constraint mechanisms that address previously identified types of separation of duty constraints.

The remainder of the paper is organised as follows. We present a summary of separation of duty constraints from the literature in Section 2, including a description of our newly identified constraint. In Section 3 we present an overview of the DTAC model and define some relationships using our mathematical approach, for which we provide a graphical example in Section 4. In Section 5 we investigate some algorithmic concerns regarding correctness and the complexity of evaluating constraints and inheritance. The problems of shared rights in mutually exclusive types and their use to implement the previously identified separation of duty constraints are covered in Sections 6 and 7. In Section 8 we compare our work to some other approaches, before presenting some concluding remarks in Section 9.

2 A Taxonomy of Separation of Duty Constraints

One of the important themes that resonates through (and predates) the literature on RBAC is separation-of-duty or conflict-of-interest constraints [SS75, SCFY96, Kuh97, SZ97, TP98, NO99, SBM99, LS99, for example]. For the sake of clarity we will present a harmonised merge of the taxonomies by Simon & Zurko [SZ97] and the extensions by Nyanchama & Osborn [NO99], adding one new constraint ourselves.

Kuhn [Kuh97] 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, but we demonstrate that our DTAC relationships naturally express these concepts in Section 6.

In the standard RBAC language [SCFY96] the harmonised 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 being assigned to the same person at the same time.

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 [BK85] and a potential part of well formed transactions [CW87].

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 [CW87].

2.1 Separation of Duty for related users

Nyanchama & Osborn identified the user–user conflict listed above. We have identified a generalisation: *separation of duty for related users.*

The goal of separation of duty is to improve security by forcing two or more users to collude to perform some illegal activity. With groups of users (such as family members) where the ties between the users are very strong the risk of collusion *appears* higher; in these cases we may wish to exclude those users from performing in mutually exclusive roles.

Thus where user–user conflicts disallow one role from being assigned to multiple users from some specified group, separation of duty for related users disallows multiple roles being separately assigned to members of the specified group of users if those roles would be disallowed from being assigned to any single user in the group.

We illustrate with two simple pictures, in Figure 1 we show two unrelated users legally inheriting from two mutually exclusive roles, while in Figure 2 we show one group (of two related users) illegally inheriting from two mutually exclusive roles.

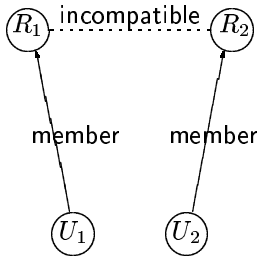

Figure 1: Separate users are allowed to be members of exclusive roles.

We will show how to enforce all these constraints in Section 7 after we have defined the necessary relationships.

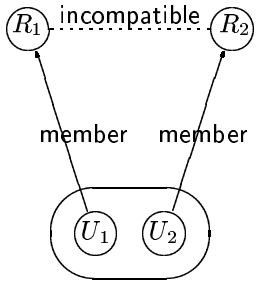

Figure 2: Illegal dual membership of exclusive roles by related users.

3 The DTAC Model

We present the DTAC model in a series of stages: first we summarise the basic DTAC model (which has no general constraints or inheritance), then we examine the mathematical basis for defining relationships (such as constraints and inheritance) before proceeding to define inheritance and a few simple constraints.

3.1 DTAC Summary

This paper defines relationships in the DTAC framework, so we feel it necessary to briefly review the conceptual pieces of the framework.

The dynamically typed access control (DTAC) model is defined [TP98] by extending the (static) type enforcement model of Boebert and Kain [BK85] to make almost all aspects of the model dynamic, in particular the typing rules. In type enforcement subjects are grouped into security types representing the subsystem to which they belong; and objects are grouped into security types which encode both the format and trustworthiness of the information contained within the objects.

Unlike many access control models DTAC does not distinguish between subjects and objects. While we expect many implementations will make the distinction for pragmatic reasons it is not a universal distinction: this distinction is simply not valid in some implementations, such as component based systems. Furthermore they are frequently ignored when modellers find it useful to consider objects as subjects [HRU76, San96].

In DTAC, types have dual security roles, which is not always obvious on an initial reading. Firstly types are used to represent general abstract security contexts, such as type enforcement domains, lattice labels or user identities. Secondly types are used to capture the access control aspect of their interface; the rights defined for an entity are a projection of the operations defined in the abstract data type of which the entity is an instance. So, for example, start/stop rights may be defined for entities implementing a thread interface but not for a file system.

Thus a basic DTAC model configuration consists of a set of entities, a set of security types, a many-to-one mapping from entities to types and a set of rights

defined by the allowed operations on entities and a permissions matrix which holds the rights entities of each type has to entities of other types based on the types of entities.

3.1.1 Translating RBAC into DTAC

Most of the related work, and in particular the taxonomy of separation of duty constraints, is defined in RBAC terminology so we present a brief mapping of RBAC into DTAC.

In DTAC all security descriptors are modelled as types; therefore *users*, *groups of users*, and *roles* are all types. This makes expressing many kinds of constraints easier because they all resolve to type-type constraints, whether they were initially role-role, user-role, user-user or user-group or group-role. One side effect of the graphical aspect of the relationships of our DTAC model is that relationships define a partial order on types; this helps resolve conflicts and improve efficiency (see section 5) but alternative constructions must be found for relationships whose natural construction runs against the order.

We also use types to model task-derived security descriptors, such as stages of assured pipelines [BK85] or well formed transactions [CW87]. Thus many task based separation of duty constraints are also easily modelled in DTAC. While it is possible to construct types to capture the long term history of entities, doing so results in a largely uncontrolled proliferation of types. Additional type structuring techniques that may or may not address this problem are under examination.

The mapping of RBAC into DTAC seems trivial, so it is sometimes easy to overlook the distinction between types (sets of entities) and roles (sets of rights to objects). The distinction is important because it is much easier to model task based security models using DTAC types than RBAC roles.

3.2 Mathematical Basis for DTAC Relationships

The mathematical basis for DTAC is sets, so it seems entirely natural to use set theory as a basis for both inheritance and constraints in DTAC. One advantage of consciously using set theory for both inheritance and constraints, is the unification of application of constraints to inheritance and of inheritance to constraints.

We do not wish to present a complete review of set theory, readers desiring a more comprehensive review of set theory and propositional logic are referred to [BS97, p501] or any introductory work on set theory for more information. Nonetheless we introduce all the constructive operators that we will use for "inheritance" and the comparative relationships that we will use to define constraints.

We classify as *operators* all those operations that can be used to change the configuration of a DTAC model, and we classify as *comparators* all those operations which cannot be used to change the DTAC configuration but which are used to construct constraints on the allowed states of the configuration.

3.2.1 Operators

There are four constructive operators that take two sets as input and produce a new set as a result:

union $A \cup B \stackrel{\text{def}}{=} \{x | x \in A \vee x \in B\}$

difference $A \setminus B \stackrel{\text{def}}{=} \{x | x \in A \wedge x \notin B\}$

intersection $A \cap B \stackrel{\text{def}}{=} A \setminus (A \setminus B) \equiv \{x | x \in A \wedge x \in B\}$

symmetric difference

$$A \triangle B \stackrel{\text{def}}{=} (A \setminus B) \cup (B \setminus A) \equiv \{x | (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$$

Collectively these operators allow us to define all eight possible sets that can be constructed from two input sets with at most one operator: $\emptyset$, A, B, $A \setminus B$, $B \setminus A$, $A \cap B$, $A \triangle B$, and $A \cup B$.

3.2.2 Comparators

There are several different ways of defining comparators, or comparative relationships, in set theory, so we will just give a definition of the ones we wish to use:

Two sets are disjoint if they have no elements in common. $A \perp B \stackrel{\text{def}}{=} A \cap B = \emptyset$

Two sets are *incomparable* if they have (or may have) some elements in common, but neither is a subset of the other.

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

Two sets are *independent* if we cannot, or choose not, to make any comparison.

The operation of finding the cardinality of a set cannot be used to define a new set, but we can compare the cardinality of a set to some specified natural number. Therefore we use we use it to define a *cardinality* comparator which tests whether a set has a particular cardinality.

3.3 Some Relationships in DTAC

The goal is to construct relationships (inheritance and constraints) from simple mathematical principles so that their interaction is well founded and easily understood.

There are two parts to defining a relationship: (1) on what is the relationship defined, and (2) how is the relationship defined.

We can define relationships on each of the three *aspects* of DTAC types: (1) the mapping of entities to types; (2) the presence (or absence) of rights between types; and (3) the presence (or absence) of relationships between these types. Some relationships are only defined on one aspect, some are defined

on multiple aspects, while some are only defined on all aspects at once; where confusion can arise as to which aspect(s) a relationship is defined on we subscript the relationship symbol with information identifying the aspect (e.g $\perp_{\cup}$) for all aspects other than rights. In particular we presume that distinct constraints will be defined that distinguish between constraints and constructive relationships, such as inheritance.

Relationships on DTAC types are defined using (1) set theory, and (2) a simple lexical substitution based on subtypes. The use of substitution in inheritance allows us to apply constraints defined on the parent (supertype) to the child (subtype).

Space does not permit us to provide a full taxonomy of relationships in DTAC, so we will focus on those relationships necessary for our examples using separation of duty and largely ignore other relationships, and the classification of those relationships we do present. In total there are four constraints and two constructive relationships:

3.3.1 Cardinality Constraints

Cardinality constraints are simple constraints on the size of a set. We can apply them separately to the number of entities mapped to a type, to the number of constraints applied to a type, or to the number of constructive (inheritance or substraction) relationships applied to a type.

The cardinality constraint of a set A is written $|A| = n$ for some natural number n, but when we draw it graphically we will typically just label the edge with the number n.

3.3.2 Disjoint Constraints

We can define two types as being disjoint in the following aspects we identified above: the existence of rights; the existence of constructive relationships; existence of other constraints; or some combination of these aspects.

If two types A and B are disjoint we write $A \perp_x B$, and we label an edge in the graph from A to B with the symbol $\perp_x$ to indicate a disjoint relationship in aspect x of the model.

3.3.3 Incomparable Constraints

Incomparable constraints were designed to implement the semantics of "shared/shared" mutual exclusion identified by Kuhn [Kuh97], which we investigate in more detail in Section 6.

We can define a pair of types to be incomparable on the same aspects as for disjoint.

Incomparable constraints are expressed the same as disjoint constraints only we use the symbol $\not\sim_x$.

3.3.4 Incompatibility Constraints

We give a special name, *incompatability*, to the constraint between two types that is the combination of an incomparable relationship on rights and a disjoint relationship on constructive (substraction and inheritance) relationships. Mathematically incompatibility is defined:

$A \not\parallel B \stackrel{\text{def}}{=} (A \not\sim B) \wedge (A \perp_{\cup} B) \wedge (A \perp_{\setminus} B)$

We use the incompatibility constraint as the primary constraint to enforce separation of duty. Several similar alternative constraints exist which could be used for separation of duty constraints. We choose this one because it appeared to offer the most flexibility without sacrificing control. However it is trivial to define new constraints as needed.

Incompatibility constraints are also expressed the same as disjoint constraints only we use the symbol $\not\parallel$.

3.3.5 Inheritance Relationships

Where RBAC defines inheritance to be a transitive set union operator on rights we define it to be a set union on rights and application of all the relationships (constraints and constructive relationships) of the parent on the child as specified by substituting the child for the parent in the constraint. We examine the algorithms for evaluating inheritance of relationships in Section 5.

We indicate inheritance with an arrow from parent to child, or supertype too subtype. Occasionally it is necessary to label an inheritance edge or to identify some constraint as operating on the inheritance aspect of types, in which case we use the set union symbol $\cup$.

3.3.6 Subtraction (rights exclusion) Relationships

It is possible to build up the set of rights a type needs, but it is frequently more convenient, and typically more concise, to construct types by inheriting from another well known type and excluding some rights. Therefore we define a substraction relationship using the set difference operator on rights.

We use the set difference symbol ($\setminus$) for subtraction, if at some later date we allow subtraction of relationships we will subscript the symbol appropriately.

4 A Graphical Example

As an example of integrated inheritance and constraints we present techniques to enforce four kinds of mutual exclusion constraints. Two of these rely on inheriting a simple constraint, one relies on inheriting a constraint on inheritance, and the fourth inherits a subtraction relationship that removes potentially conflicting rights.

Readers not comfortable with switching between RBAC and DTAC terminology may wish to refer to Section 3.1.1.

In Figure 3 we show a simple inheritance graph where solid arrows are used to represent inheritance relationships (from parent to child) over both rights and constraints. In RBAC, if user U is a member of role R then an arrow is drawn from U to R, and U inherits the rights of R; whereas in DTAC the corresponding inheritance edge is drawn directly from R to U.

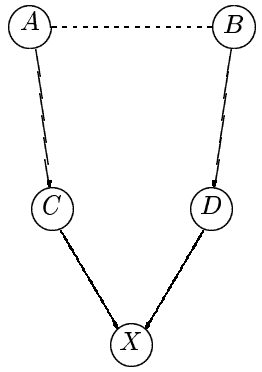

Figure 3: An inheritance graph amoungst DTAC types.

There are then four ways that we can define mutual exclusion between types A and B and their descendants:

1. By placing a disjoint rights relationship between type A and B:

   $$A \perp B$$

   Attempting to define X as shown would have X inherit the rights and constraints of both types A and B (via C and D respectively) which is contradictory — X would have A's rights by inheritance via C and be defined to have no rights in common with A via inheritance from B.

   Therefore no type may be defined that inherits rights from both A and B.

2. By placing an incomparable relationship between types A and B:

   $$A \not\sim B$$

   Attempting to define X as shown would have X inherit the rights and constraints of both A and B (via C and D respectively) which is contradictory — anything inheriting A's (B's) rights would have a superset of A's (B's) rights and clearly be comparable to A (B).

3. By placing an incompatibility relationship between types A and B:

   $$A \not\parallel B$$

Any type inheriting from A is constrained by the incompatability relationship to have no inheritance relationships with B, and vice versa. Therefore it is illegal to try to define any type that inherits from both A and B, independent of the rights held by any of the types; thus unlike incomparable constraints incompatibility constraints are enforced even if a later substraction removes conflicting rights or they are never added.

4. By placing subtraction relationships between A and D and between B and C:

$$C' \leftarrow C \setminus B\ , \quad D' \leftarrow D \setminus A$$

The subtraction relationship explicitly removes from C and its descendants any rights held by B, and from D and its descendants any rights held by A. The result is mutual exclusion, so long as the rights are not directly added to a descendant type.

In these examples we only specified constraints within a fragment of the whole type graph. Using our integrated model of inheritance and constraints we can easily extend these to cover the whole graph by applying constraints between these types (A and B) and the roots of the other subgraphs. Inheritance will propagate the constraints throughout the rest of the graph. This is a major benefit of the integrated approach to inheritance and constraints.

5 Algorithmic Concerns

A naive algorithm for calculating the rights possessed by a particular type works like this:

1. start with the type we wish to calculate the rights for
2. construct the inheritance antecedents of the type by recursively following inheritance links in reverse
3. starting at the top of the constructed inheritance graph for this type, evaluate each relationship (at this point treat inheritance relationships as set unions on rights) lexically replacing all occurrences of the type currently being examined with the type we wish to calculate rights for

This algorithm is highly inefficient, and we will examine some significant efficiencies that flow from imposing a few restrictions on arbitrary relationships.

There are four simple restrictions on the order of the combination of the effects of multiple relationships: (1) relationships defined using set union must be evaluated before relationships defined using set difference; (2) the constructive relationships must be evaluated before the constraints; (3) all the relationships

must be evaluated for one type before relationships on other types that referencing the first type; and (4) that inherited relationships are evaluated in the same order that the types they were inherited from were evaluated.

These requirements effectively define a partial order on types, which we do for three reasons:

1. the cyclic definition of types using constructive relationships is not sound;
2. the sensible resolution of conflicts resulting from multiple relationships requires some ordering; and
3. there are well known efficient algorithms for working with directed acyclic graphs.

Our partial ordering of types makes it easy to natural the *path override capability* and *denials over grants* policies for conflict resolution [BJSS97, JSS97]. This policy favors allowing denial-of-service attacks (by denying too much) over breaches of integrity and confidentiality (by allowing too much). We feel this is justified because denial of service attacks are more likely to brought to an administrators attention, and thus can thus be addressed, sooner than breaches of integrity and confidentiality which are likely to go unnoticed even if they are not actively hidden.

Dynamic programming is the main approach for improving the performance of the naive algorithm described above. With dynamic programming, DAG traversal algorithms have worst case complexity that is quadratic in the number of nodes in the graph (number of types). However the worst case only occurs in what is recognisably a poorly structured set of types — a completely backwards connected linear sequence of types. In general we believe that in a well structured set of types, most types will inherit from only one or two other types; in this case the complexity will approach linear-log time. Furthermore the algorithm only needs to examine the subtrees rooted at the types that changed; we believe that an additional characteristics of a well designed tree of types is that core types will be near the root of the graph and will change much less often than supplementary types which will form the leaves.

6 Shared Rights in Mutual Exclusion

This section focuses on the question of sharing rights between mutually exclusive roles that was identified by Kuhn [Kuh97] as one of the important criteria in defining mutual exclusion. We believe this question goes to the heart of the definition of separation of duty, though it appears to have largely ignored by the RBAC community.

The question has two parts:

A) what rights may be shared between mutually exclusive roles? and

B) what rights may be shared between roles that are part of some specified mutual exclusion and other roles in the system?

While Kuhn did not clearly state this, the question can only be asked relative to some fixed set of objects. We demonstrate this with a simple example. Consider two mutually exclusive roles that have security relevant implications: to enable these roles to generate audit trail information they must both have append access to an audit trail object. Clearly these roles are not mutually exclusive with respect to the audit trail.

Let us explore this example in more detail to show how we implement it in DTAC. If we define two types T_1 and T_2 which we wish to be mutually exclusive with respect to some other types, but we wish both T_1 and T_2 to have access to the audit log which is granted by inheriting from type A. Then we have the two situations shown in Figure 4 in which T_1 and T_2 are constrained to be *disjoint*, and Figure 5 in which T_1 and T_2 are constrained to be *incomparable*. The descendent types T_1' and T_2' are now incomparable ($T_1' \not\sim T_2'$) to each other, but $T_1' \perp T_2$ and $T_1 \perp T_2'$. So the semantics of the separation requirement is upheld. This illustrates that while we can choose to define mutual exclusion as a disjoint constraint with regard to other types (all types T_1 and T_2 have rights to) the types which will most often be used are are simply incomparable ($T_1' \not\sim T_2'$).

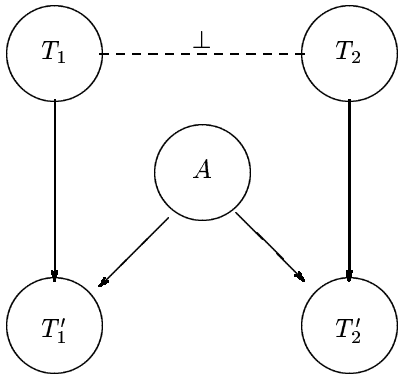

Figure 4: Inheritance of mutual exclusion enforced with a disjoint relationship.

With the richer set of relationships in the DTAC model we can define two additional questions:

C) what relationships may be shared between mutually exclusive roles ? and

D) what relationships may be shared between roles that are part of some specified mutual exclusion and other roles in the system ?

Another example is that a pair of mutually disjoint roles may share a *disjoint* relationship with a third role with which they are disjoint, though they may not share an *inheritance* relationship with some fourth role as they would no longer be disjoint.

In our examples (Section 4) we presented four different ways of enforcing mutual exclusion. These identify two types of mutual exclusion that did not occur in Kuhn's model which lacked such relationships.

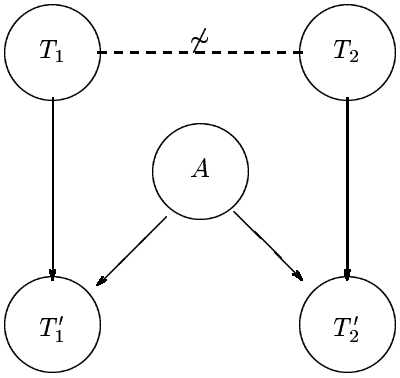

Figure 5: Inheritance of mutual exclusion enforced with an incomparable relationship.

7 Implementing Separation of Duty Constraints

In this section we wish to show how to specify the separation of duty constraints of Section 2 using the DTAC relationships of Section 3.3. The constraints were originally defined without inheritance, to demonstrate the simple expressiveness of relationships in DTAC. We specify these constraints using inheritance to provide structure and simplify the specification.

When they defined their taxonomies, Simon & Zurko and Nyanchama & Osborn, did not include aspects of shared rights (see section 6 above) so we will take the "shared/shared" approach of Kuhn. The "shared/shared" approach explicitly allows some but not all rights to be shared between mutually exclusive roles — it is exactly what the *incomparable* relationship was defined to implement. Thus we define separation of duty constraints using *incomparable* and *incompatible* relationships bound together using *inheritance* relationships.

We begin by showing how to specify a simple separation of duty constraint, and then how to specify our *related user separation of duty* constraint (see Section 2.1). Finally we sketch the outlines of the solutions for the other constraints.

A *simple dynamic separation of duty* constraint disallows two particular roles being assigned to the same person at the same time.

This is easily implemented by placing an *incompatible* relationship between the pair types representing the mutually exclusive roles, see Figure 7. It is then illegal for any type to inherit from both of the types (as it will inherit from each an incompatibility with the other from which it is also inheriting).

Then when users are assigned to either type by *inheritance* they automatically inherit the exclusion with the second role for as long as they are assigned the first role.

Our *related individual separation of duty* constraint is implemented using the same technique, but now a new type is created which inherits from all the related users, as shown in Figure 6. To ensure that the type representing the group of related users is never (mis)used we place a zero cardinality constraint

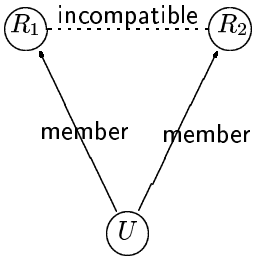

Figure 6: Illegal dual membership of exclusive roles.

on entities assigned to the group type.

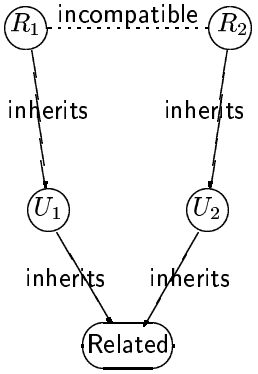

Figure 7: Implementing sepation of duty for related users.

Most of the remaining separation of duty constraints are implemented using similar techniques. We provide sketches of those techniques below.

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

This is easily implemented by placing an *incompatible* relationship between the type representing the user and the type representing the disallowed role when the user is first assigned to one of the mutually exclusive roles. Thus the user can never inherit rights from the mutually exclusive role, whether or not they give up the first role.

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.

This is easily implemented by creating two types which are each assigned one of the sets of rights from the conflicting privileges, and then defining either a static or a dynamic separation of duty constraint between these types.

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

This is implemented using an *incompatible* relationship between the type representing the user and the types representing the roles the user may not be assigned to.

User–user conflicts are defined to exist if a pair of users should not be assigned to the same role.

This is implemented via a disjoint constraint on the inheritance relationship of the users $U_1 \perp_{\cup} U2$. Thus, the inheritance relationships between the users must be disjoint.

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 for a single business task.

This can be implemented by defining a type to represent the task as a whole and assigning types for each stages to this task type. By placing a cardinality constraint (on inheritance from the task type) of at most 1 less than the number of stages on the task type, any user that tries to be assigned to too many stage types will run afoul of the cardinality constraint.

Object-based separation of duty constrains a user never to act on the same object twice.

This constraint should be enforced by adding disjoint relationship between the user and the object upon use. This prevents the user from inheriting the restricted rights to the object. Clearly, the number of these relationships can increase rapidly, but we view this as an unusual constraint.

Order-dependent History constraints restrict operations on business tasks based on a predefined order in which actions may be taken.

This is an assured pipeline [BK85] and can be implemented by defining a type for each stage and changing the types of entities as they progress through the stages. This constraint is easy to implement because DTAC is a series of extensions to type enforcement [BK85] which was designed to implement assured pipelines.

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.

This can be implemented by creating multiple interwoven assured pipelines to represent every possible order that the actions may be performed in.

8 Related Work

The most commonly used RBAC models (as identified by references from the ACM RBAC workshop proceedings [RBA95, RBA97, RBA98, RBA99]) define roles to be collections of privileges, and privileges to be a pair $rights \times objects$. Using this definition there is no facility for the object labelling necessary for several of the object and task based separation of duty constraints, such as *operational separation of duty*. Therefore it is not possible in these models to express all the separation of duty constraints.

Constraints have been part of most RBAC models of recent years [SCFY96, NO99, SBM99, LS97, BFA99, BJSS97] but with the exception of Nyanchama & Osborn they have always been specified with a rule-based systems (previous papers about DTAC [TP98, TOP99] used graph theory to bound the complexity of constraint solving but did not include details). Unfortunately rule-based systems, while highly expressive are harder to visualise and thus to use; thus they are frequently avoided by non-experts, or worse (because it creates a false sense of security), they are incorrectly specified. A fact evidenced by the lack of commercial RBAC implementations that include a rule-based constraint model. Although we have previously advocated a rules based approach to specifying constraint [TP97] because we identified safe aspects of change control that cannot be effectively expressed in simple tables or graphs, we now advocate using a graphical approach as it is more important to get the common separation of duty constraints correctly than to allow some esoteric but safe dynamism.

Nyanchama's & Osborn's role-graph model of RBAC is the most similar to our work: they have a simple graphical model for role-role relationships based on a lattice which includes a combined view of role inheritance and those separation of duty constraints that can be expressed within the role-graph. Some constraints, such as those involving users, cannot be expressed entirely within the role-lattice. As we have demonstrated it is possible to construct graphical representations for most these, though it is not clear whether Nyanchama & Osborn have done so.

In every RBAC model that we are aware of, inheritance is defined by subset inclusion of the privileges of the parent role in the privileges of the descendant role [Bal90, SCFY96, LS99, NO99, for example]. As simple as this subtype preserving definition is, there are several problems with it. Firstly, there is no reason to limit inheritance to roles — it can be usefully applied to rights and objects as well [TOP99], and, as we have now shown, to other relationships such as constraints. Secondly, many real world scenarios require role hierarchies which do not obey a strict subtype relationship [Mof98, ML99] so that inheritance structures are less useful than might be hoped. Thirdly inheritance defines only an additive way of constructing subtypes, and does not provide an intuitive way of defining exceptions.

We identify three specific problems in the related work that our work addresses:

1. the typed objects of DTAC allow us to express constraints that cannot be

expressed using privileges based on a $rights \times objects$ pair;

2. we have a simple graphical representation of more types of constraints than any previous work constraints; and

3. by constructing constraints and inheritance from mathematical principles we have a more integrated and coherent picture of their interaction.

A major benefit of our constructing relationships from mathematical principles is that we have identified new kinds of relationships that are naturally included in a graphical model. These new relationships directly contribute to the increased simplicity and expressivity of our model over previous work.

9 Conclusions and Future Work

We set out to develop a simple integrated model for inheritance and constraints in DTAC. Basing our design on a mathematical foundation resulted in an integrated model of inheritance and constraints. The integrated model allows a simpler and more expressive graphical representation than previous graphical models.

In the process of developing our model we have made some simple contributions to general RBAC modelling: (1) we have identified a new constraint, (2) we have defined new operators, and (3) we have shown that a simple graphical model of constraints is sufficient to accomplish the most common constraints.

We are currently implementing a prototype that uses this model to implement access control for componentized system services. In the future, we expect to both gain practical experience using the model as well as develop a complete theoretical model based on these results and further analysis.

References

[Bal90] R. W. Baldwin. Naming and group priviliges to simplify security management in large databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1990.

[BFA99] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information System Security*, 1(2), February 1999.

[BJSS97] 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.

[BK85] 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.

[BS97] I. N. Bronshtein and K. A. Semendyayvev. *Handbook of Mathematics.* Springer (Berlin), 3 edition, 1997. English translation edited by K. A. Hirsch.

[CW87] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceeding of the IEEE Symposium on Security and Privacy*, Oakland, California, 1987.

[HRU76] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[JSS97] 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.

[Kuh97] 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.

[LS97] 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.

[LS99] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 1999. To appear.

[ML99] J. D. Moffet and E. C. Lupu. The Uses of Role Hierarchies in Access Control. In *Proceedings of 4^{th} ACM Workshop on Role-Based Access Control*, October 1999.

[Mof98] J. D. Moffet. Control Principles and Role Hierarchies. In *Proceedings of 3^{rd} ACM Workshop on Role-Based Access Control*, November 1998.

[NO99] 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.

[RBA95] Proceedings of the 1^{st} Workshop on Role-Based Access Control, November 1995.

[RBA97] Proceedings of the 2^{nd} Workshop on Role-Based Access Control, November 1997.

[RBA98] Proceedings of the 3^{rd} ACM Workshop on Role-Based Access Control, October 1998.

[RBA99] Proceedings of the 4^{th} ACM Workshop on Role-Based Access Control, October 1999.

[San96] R. S. Sandhu. Role Hierarchies and Constraints for Lattice Based Access Controls. In *Proceedings of 4^{th} European Symposium on Research in Computer Security*, September 1996.

[SBM99] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information System Security*, 1(2), February 1999.

[SCFY96] R. S. Sandhu, E. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Acess Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[SS75] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), September 1975.

[SZ97] 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.

[TOP99] J. E. Tidswell, G. H. Outhred, and J. M. Potter. Dynamic rights: Safe extensible access control. In *Proceedings of the 4^{th} ACM Role-Based Access Control Workshop*, October 1999.

[TP97] J. Tidswell and J. Potter. An Approach to Dynamic Domain and Type Enforcement. In *Proceedings of the Second Australasian Conference on Information Security and Privacy*, July 1997.

[TP98] J. Tidswell and J. Potter. A Dynamically Typed Access Control Model. In *Proceedings of the Third Australasian Conference on Information Security and Privacy*, July 1998.