CatBAC: A Generic Framework for Designing and Validating Hybrid Access Control Models

Bernard Stepien† Hemanth Khambhammettu‡ Kamel Adi‡
Luigi Logrippo‡

†Department of Computer Science and Engineering, University of Ottawa, Canada

‡Laboratoire de Recherche en Sécurité Informatique, Université du Québec en Outaouais, Canada

Email: bernard@eecs.uottawa.ca, {hemanth.khambhammettu, kamel.adi, luigi.logrippo}@uqo.ca

Abstract

Many access control models have been proposed in the literature, and they have been extensively studied under the acronyms of DAC, MAC, RBAC, ABAC, etc. Each of these models has been studied in isolation, but some real-life situations need elements of several of them, in order to properly express data protection needs of complex organizations. A formal framework is presented, that allows not only to combine elements of these models, but also to generalize them in new ways. This framework includes elements of a lifecycle methodology, which starts with a UML-based formalism, called UACML, that expresses semantic elements (classes and their relationships) needed for general access control systems. It continues with the representation of UACML diagrams in our language CatBAC. The latter is a compact textual representation of UACML that makes it possible to express realistic policy systems involving many entities and many constraints. CatBAC is based on Prolog, and this makes it possible to implement analysis and verification tools.

1 Introduction

Access control requirements are an important part of overall system security requirements. The various access control languages that exist today are the result of a constant evolution in security requirements. Over the last four decades, various approaches to access control have been developed, among which mandatory access control (MAC), discretionary access control (DAC), role based access control (RBAC) and attribute based access control (ABAC) are well known. Each of the existing access control languages and models has its advantages and limitations.

Traditionally, designers of access control policies choose one of the existing access control models and corresponding languages that best fit the security requirements of their applications. MAC models may be used for military systems, whereas DAC, RBAC or ABAC models may be chosen for banks or hospitals. However, access control requirements of organizations are becoming increasingly complex, sometimes making access control mechanisms based on a particular access control model either inefficient or inappropriate: combinations of access control models may then be needed. We refer to models which are based on combinations of two or more access control models as "hybrid" models.

Naturally, hybrid access control models are more complex than access control models which are based on a given single access control paradigm, leading to the possibility of errors during their specification or enforcement. It is well known that validating the requirements during early phases of the software development life cycle reduces the generation and propagation of errors and cost of fixing those errors during later phases. Hence, there is a need for a disciplined "life-cycle" approach towards hybrid access control policies, which can reduce the possibility of errors starting from the early phases of design. In particular, design disciplines using visual modeling during the early stages, and logical verification during the later stages, would be useful for reducing the incidence of errors.

Towards this end, our previous work [1] has resulted in the development of a UML-based unified access control modeling language (UACML) that is a modeling framework for the visual design of hybrid access control policies. In this paper, we provide the concepts for a textual language that corresponds to UACML and show how to translate the access control requirements expressed with UACML into this textual language. We then present a Prolog-based verification tool for detecting errors within the textual language, which may have been propagated from UACML design models. We envisage that these errors will be addressed by appropriately modifying the UACML graphs, and then repeating the translation and verification cycles until a presumably correct specification will have been obtained.

The rest of the paper is organized as follows. Section 2 briefly describes our previous work of UACML that provides support for visual modeling of access control policies. In Section 3, we translate the UACML visual graphs into a textual language. Section 4 describes a Prolog-based verification technique for the textual language developed in Section 3. Section 5 compares the work presented in this paper with relevant work of the literature. We draw conclusions for this paper in Section 6.

2 Overview of UACML

UACML [1] is a UML-based modeling language for visualizing hybrid access control policies and is based on an extension of the metamodel of access control proposed by Barker [2]. The central component of UACML is the notion of a *category* that is instrumental to abstract key components of various access control models, such as groups, roles and security levels. Essentially, categories provide means to associate subjects and permissions, which are represented as `<resource,action>` pairs. Note that the class Category is defined as an abstract class (represented in Fig. 1 by italicized font). Hence, the class Category must be specialized for creating specific categories, such as groups, roles and security levels, to which subjects or resources can be assigned.

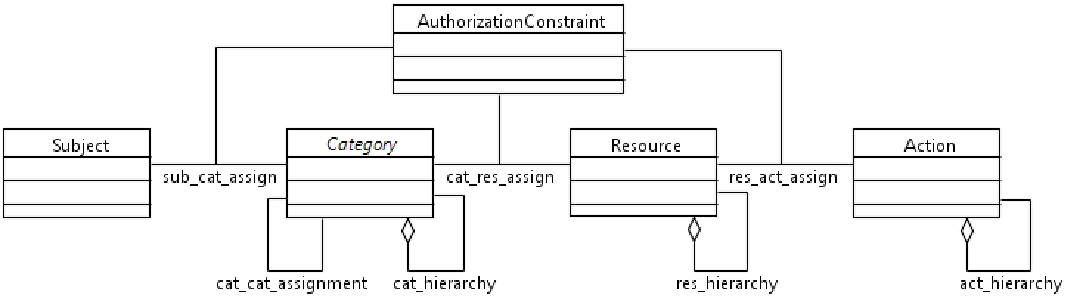

Figure 1: UACML Meta-metamodel

The left-side of class Category shows a class Subject that represents entities within the system that are able to initiate access request(s). The class Subject has an association with class Category. The right-side of class Category shows two classes: Resource and Action. The class Resource represents protected resources within the system and is associated with the class Category. Such an assignment is useful to represent which resources are accessible by a given category.

The class Action represents operations that can be performed on protected resources. The class Resource has an association with class Action; thus, representing which actions can be performed on given resources.

UACML provides support for hybrid access control policies by allowing categories to be associated with other categories (represented in Fig. 1 as a self-association edge on class Category). Furthermore, UACML allows the specification of hierarchical relationships between categories by aggregating appropriate categories (represented in Fig. 1 as a self-association edge with a diamond head on class Category). The metamodel of UACML also provides support for creating resource hierarchy and action hierarchy by aggregating resources and actions, respectively. Such hierarchical relationships are represented by an aggregation association (depicted in Fig. 1 as an association edge with a diamond head) on classes Resource and Action. Finally, the class AuthorizationConstraint is used to represent constraints that specify restrictions on subject-category assignments, category-resource assignments and resource-action assignments.

Fig. 2 illustrates the specialization of abstract class Category into classes Group, Role and SecurityLevel. Note that the metamodel shown in Fig. 2 illustrates the specification of hybrid access control policies, which require associations between different types of specialized categories (Group, Role, SecurityLevel, for example).

3 Category Based Access Control

3.1 The need for a textual language

While graphical information can be easier to understand than textual information, this property can reverse itself as the amount of information increases. The graphical UML

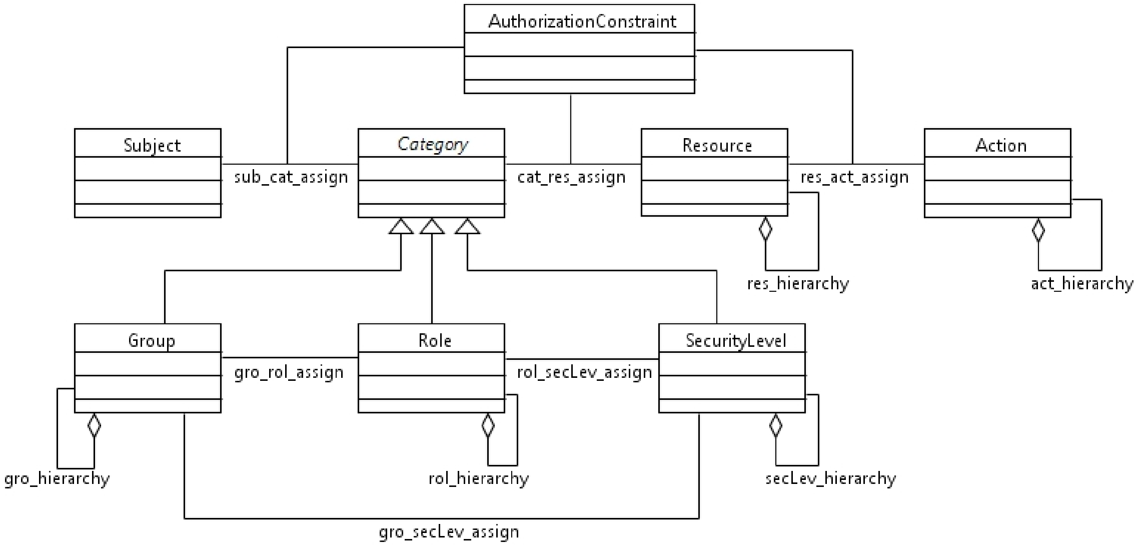

Figure 2: Generating metamodels

like notation of UACML can be transposed easily to a structured textual language. For this purpose, we have developed a ”strongly typed” user-friendly language that forces a discipline on the use of certain predefined constructs which rely on a clearly defined grammar and appropriate keywords in order to avoid ambiguities. We call this textual language Category Based Access Control (CatBAC).

3.2 CatBAC language constructs

A distinct language construct is used for each kind of assignment, which can be of the following three kinds:

- Assignments of subjects to categories
- Assignments of categories to other categories
- Assignments of categories to resources/actions

An intuitive description of the main CatBAC characteristics and constructs follows, where CatBAC keywords are shown in bold.

3.2.1 Strong typing

In order to minimize run-time errors, it is useful to use strong typing. Thus, categories, resources and actions should be defined in type declarations as enumerations as shown below:

type resources enumeration `RFP, input_RFP, resp_RFP, bid_RFP;`

```
type actions enumeration read, write;
type categories enumeration role, group, security_level;
```

This principle can be extended to values inside one of the above elements such as:

```
type category group enumeration {Project_1,Project_1A, Project_1B}
type category role enumeration {employee, manager,consultant, contractor}
```

3.2.2 Assignment of subjects to categories

A typical RBAC role assignment reads in CatBAC as follows:

```
assign subject Alice to role Consultant;
```

However, with CatBAC, a subject can be assigned to a category that is not a role. In the following example, it is a group:

```
assign subject Alice to group Project_1A;
```

The assignment can also be to a security level as follows:

```
assign subject Alice to security_level Unclassified;
```

It is important to note that there can be multiple assignments of subjects to various categories simultaneously. This is an extension of the multiple assignments to roles that are possible in the RBAC model. Thus, in the three previous examples, subject Alice can be specified concurrently as shown in Fig. 3.

3.2.3 Assignments between categories

The second kind of assignment is between categories themselves. For example, a category called group could be assigned to a category security level. This enables considerable conciseness in specifying access control policies.

```
assign category group Project_1B to category security_level Classified;
```

3.2.4 Assignments to resources/actions

The assignment of permissions to subjects or categories to `<resource,action>` pairs is a special kind of assignment and can be summarized in a single command such as:

```
assign permission permit to category role Manager for resource Bid_RFP
                                                        and action Write;
```

Here, the textual notation is more expressive than the UACML notation because in a single statement it associates both resources and actions which were represented in UACML as two separate sets of nodes and edges. We can also achieve conciseness by assigning a permission to several categories, resources and actions in a single statement as follows:

```
assign permission permit to categories role Consultant, Manager
                  for resources Input_RFP, Bid_RFP and actions read, write;
```

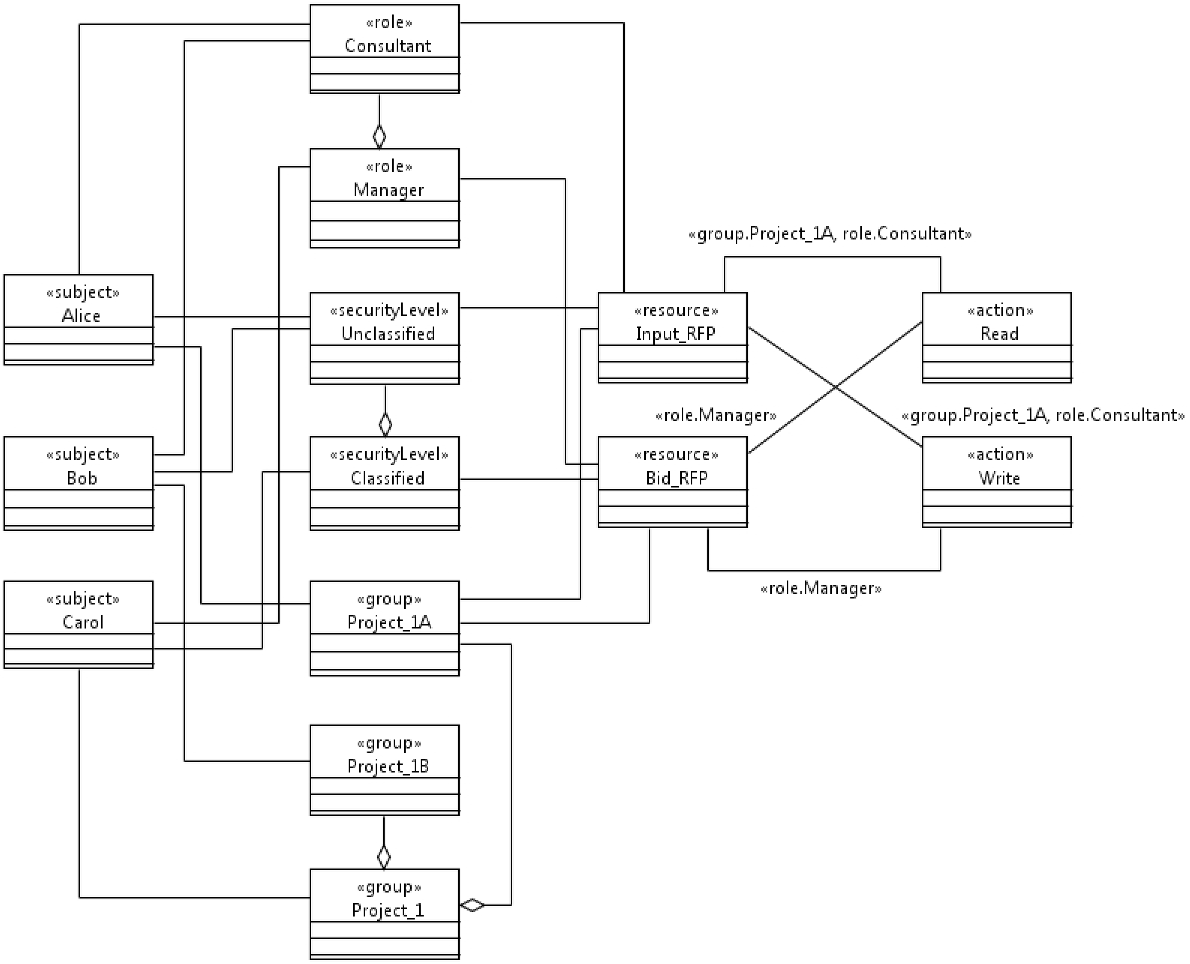

Figure 3: A hybrid model

3.2.5 Representing mandatory assignments

Obligations require subjects to possess certain attributes or perform some actions [3]. Obligation constraints really correspond to the concept of mandatory assignments to permissions. For this purpose, we use the keyword **mandatory** following immediately the keyword **assign** as follows:

assign mandatory permission permit to category `group Project_1A`

for resource `Input_RFP` **and action** `Read;`

The above statement means that a subject has to be assigned directly or indirectly (through chains of category assignments) to group `Project_1A` in order to be granted access to resource `Input_RFP` and action `Read`. Several mandatory assignments are allowed.

3.2.6 Representing inheritance

The RBAC model includes the concept of role inheritance. This concept can be easily extended to categories in general. For example, the group category could be sub-divided into subgroups forming a hierarchy as shown on Fig. 3 and can be represented textually as follows:

category `group Project_1` **inherits from** `group Project_1A;`

The above example suggests a semantic difficulty: a category can inherit only from an instance of the same category. This explains why the keyword **category** is stated only once. Mixing two categories in an inheritance statement generates a compilation error due to strong typing. The same concept can be extended to resources and actions as follows:

resource `chemistry_book` **inherits from** `books;`
action `browse` **inherits from** `read;`

3.2.7 Representing constraints

UACML uses OCL for representing constraints. However, OCL has some syntactic peculiarities that may make it not very intuitive. Thus, a more natural language construct has been developed for the textual representation of constraints.

The most widely used constraints in the literature are Separation of duties, requirements and cardinality [4, 5, 6]. All these types of constraints are applicable or extendable in the CatBAC model. In RBAC, 'separation of duties' only applies to roles. In CatBAC, it is possible to specify a constraint that requires a subject cannot be assigned to two different categories. Furthermore, such Constraints can also be specified within the same category or between different categories. Given below are some examples.

Mutual exclusion constraints Since CatBAC allows categories that are not necessarily role based, the terminology 'mutual exclusion' is more appropriate than 'separation of duties'. The following is an example of mutual exclusion in the category role.

category `role teacher` **and category** `role student` **are mutually exclusive**;

The above example is equivalent to traditional RBAC separation of duties. The following example illustrates mutual exclusion specified over different kinds of categories. It says that a manager cannot belong to `group_2A`:

category `role manager` **and category** `group Project_2A` **are mutually exclusive**;

Requirement constraints Again, requirement constraints can be within a category or between distinct categories.

category assignment `role teacher` **requires category assignment** `role researcher;`

Cardinality constraints Cardinality constraints are used to restrict the number of assignments of subjects to categories, including roles and groups. We use the keywords **should not exceed**, **should be equal** and **should be over** to express the well-known comparison operators.

category `role dean` **assignments should not exceed** `1;`

4 Verification Strategy

The graphic UACML notation already suggests that both execution through a decision engine and verification can be achieved by following paths between subjects and `<resource,action>` pairs in a graph. Also, the concept of assignment can be easily represented by relational database tables. The need to reason on sets of rules can be addressed by using logic programming languages such as Prolog.

Prolog enables rapid prototyping of an execution engine, having an internal data base of facts that can be used to store policies. In addition, it has a long history of having been used for verifying RBAC models [7]. The Prolog backtracking features are an efficient way to explore alternative paths through the UACML graph which can be associated with conflicts. Although the Prolog mechanisms that we describe below may not be sufficiently efficient for sets of policies of industrial size, they are ideal for prototyping the reasoning. Efficient industrial implementation is left to future research.

4.1 Translation into Prolog

The CatBAC language constructs can be mapped on a one-to-one basis to single Prolog facts.

4.1.1 Assignments representation

Our approach consists in using the fact name *assignment* and in placing the attributes of the assignment as arguments to this fact. The same fact name is used for various kinds of assignments. The first argument indicates the kind of assignment.

The assignment of subjects to categories is represented as follows:

```
assignment(subject, Alice, role, Consultant).
```

The assignment of categories to other categories is represented as follows:

```
assignment(group, Project_1B, security_level, Classified).
```

Inheritances can be represented in a similar way using category name and value pairs as arguments.

Textual rules:

category `role Manager` **inherits from** `role Consultant;`
resource `chemistry_book` **inherits from** `books;`
action `browse` **inherits from** `read;`

Prolog facts:

```
category_inherits(role, Manager, role, Consultant).
resource_inherits(chemistry_book, books).
action_inherits(browse, Read).
```

Permissions on resources and actions combinations are represented also by a single Prolog fact as follows:

permission`(r3, role, Consultant, Input_RFP, write, permit).`

And finally, constraints are represented as follows:

```
category_exclusion('role','teacher', 'role', 'student').
```

```
category_prerequisite(‘role’, ‘teacher’, ‘role’, ‘researcher’).
category_cardinality(‘role’, ‘dean’, ‘1’).
```

4.2 Language execution

4.2.1 Execution predicates

An execution engine is used to evaluate requests in order to grant or deny access to a resource for a given action. This is commonly the role of a policy decision point (PDP). We use an `evaluate_request` predicate that takes a request composed of a subject (S), a resource (Rs) and an action (A), and returns a path of assignments between a subject and a resource and action pair.

```
evaluate_request(request(S,Rs,A),D, Path):-
   retractall(found_path),
   find_category(S, [], LS, RN, D),
   find_resource(RN, Rs, LS, RLS),
   find_action(RN, A, RLS, ALS),
   Path = solution(RN, ALS),
   nl, write(‘checking mandatory permissions’),
   check_mandatory_permissions(S, Rs, MPD, A, MPath).
...
evaluate_request(_, not_applicable, solution(‘no rule’, not_applicable)).
```

where the `find_category` predicate consists in finding an assignment between the subject and a category. When such an assignment is found, it is appended in the list that will constitute the path through the graph and then the found category will be used to verify the nature of the next link toward the resource.

```
find_category(Cat, PL, NL, RN, D):-
   assignment(FromCatKD, Cat, CatKd, ToCat),
   append(PL, [assignment(FromCatKD, Cat, CatKd, ToCat)], NNL),
   verify_category(ToCat, RN, NNL, NL, D).
```

The `verify_category` predicate first checks if there exists a link to a permission specification. If not, it backtracks and checks if there is a category inheritance and finally in the case of failure it will check if there is an assignment to another category. The last case would indicate a path through several category assignments such as for example an assignment between a role and a group.

```
verify_category(ToCat, RN, PL, NL, D):-
   permission(RN, Cat, ToCat, Rs, Ac, D),
   append(PL, [permission(RN, Cat, ToCat, Rs, Ac, D)], NL).

verify_category(ToCat, RN, PL, NL, D):-
   category_inherits(CatKd, ToCat, CatKd, IHCat),
   append(PL, [category_inherits(CatKd, ToCat, CatKd, IHCat)], NNL),
   verify_category(IHCat, RN, NNL, NL, D).

verify_category(Cat, RN, PL, NL, D):-
```

```
   assignment(FromCatKD, Cat, CatKd, ToCat),
   append(PL, [assignment(FromCatKD, Cat, CatKd, ToCat)], NNL),
   verify_category(ToCat, RN, NNL, NL, D).
```

The separation of categories, resources and actions assignment searches into three separate predicates `find_category`, `find_resource` and `find_actions` is an efficient way to handle categories, resource and action inheritances in the case that the requested resource has no corresponding permission fact. In the following code, the first case of the `find_resource` predicate will merely be satisfied if the requested resource matches a permission fact that holds that resource. But if such permission is not found, then the second `find_resource` predicate checks recursively if an inherited resource matches the request. The recursion enables to walk complex inheritance paths.

```
find_resource(RN, Rs, L, L):-
   permission(RN, _, _, Rs, _, _).

find_resource(RN, Rs, PL, NL):-
   resource_inherits(Rs, IHRs),
   append(PL, [resource_inherits(Rs, IHRs)], NNL),
   find_resource(RN, IHRs, NNL, NL).
```

4.2.2 Execution example

Using Fig. 3, we can place the request for subject Carol to access resource `Input_RFP` for the purpose of reading. This can be achieved via the following Prolog query:

```
?- evaluate_request(request(carol, input_RFP,read), D, Path).
```

Fig. 3 indicates subject Carol to be assigned only to role Manager and group `Project_1`. Furthermore, resource Input_RFP is assigned directly neither to role `Manager` nor to group `Project_1`. However, role `Manager` inherits from role `Consultant` that is assigned to resource `Input_RFP` and thus a path between subject Carol and resource `Input_RFP` exists for action Read. The Prolog query will thus provide the following solution path:

```
D = permit,
   Path = solution(r2,
   [assignment(subject, carol, role, manager),
   category_inherits(role, manager, role, consultant),
   permission(r2, role, consultant, input_RFP, read, permit)])
```

If we had assigned a group `Project_1A` to a permission on resource `Input_RFP` and action `Read` directly instead, we would have obtained additional solutions such as the one described above for subject Carol through role inheritance as follows:

```
D = [permit],Path = [mandatory_permission(r8, group, Project_1A, Input_RFP, read,
permit),
   category_inherits(group,Project_1, group, Project_1A),
   assignment(subject, carol, group, Project_1)]
```

4.2.3 Conflict detection

We consider the following three kinds of conflicts: permission conflicts (modal), mandatory conditions violations and constraints violations.

Permission conflicts Permissions conflicts occur when there exist several solutions that return opposite effects permit and deny. This case can result either from several assignments between subjects and categories that are themselves assigned to resources or from actions resulting in opposite effects. In Prolog, this can be verified by using the `findall` built-in predicate in conjunction with our `evaluate_request` predicate as follows:

```
findall(P,
   evaluate_request(request(carol, Input_RFP, read),D, P), LP).
```

Where variable `LP` returns the list of solutions. This list of solutions needs to be submitted to another predicate that merely checks that all solutions have returned the same effect.

Mandatory conditions violations A conflict exists if several solutions to a request include a variety of mandatory and non-mandatory assignments. This suggests that someone could be granted access to a `<resource, action>` via a path of assignments while there is no corresponding path that traverses a mandatory assignment, thus resulting in ignoring the mandatory assignment.

Constraints violations The three kinds of constraints mentioned above can be easily verified with Prolog. The verification of mutual exclusions can be merely implemented by verifying if there exist two separate paths that traverse two different assignments that appear in a mutual exclusion predicate, this regardless of whether assignments belong to the same category class.

```
verify_separation_of_categories:-
   category_exclusion(_, CatName_1,_, CatName_2),
   checkPathToSubject(SBJ, CatName_1, [], Path_1),
   checkPathToSubject(SBJ, CatName_2, [], Path_2),
   ...% display violation details
   fail.
```

5 Related work

Recent research has resulted in the development of UML-based modeling languages that incorporate security requirements into system design models [8, 9, 10, 11, 12, 13]. Epstein and Sandhu proposed a framework that uses UML notations for employing UML as a language to represent RBAC requirements [8]. Shin and Ahn proposed an alternative technique to utilize UML notation for modeling RBAC requirements [9]. Doan et al provided support for incorporating MAC into UML diagrams [10].

The work of Basin *et al* includes a security modeling language, called SecureUML, that is developed on an extension of RBAC and provides support for developing system design models which include access control requirements [11]. Ray et al proposed parameterized UML elements for modeling an access control framework that combines MAC and RBAC in order to express hybrid access control policies [13]. Similarly, the work of Pavlich-Mariscal *et al* [12] proposes a modeling language that provides support for incorporating a variety of access control requirements, such as DAC, MAC and RBAC, into security design models. Their modeling language also provides support for combining different access control models for specifying policy requirements.

Our approach comprises of logic-based programming that facilitates the automation of validation. Our framework is designed in a flexible way that allows the validation of different types access control policies. Note also that our textual language is based on a unified representation of various access control paradigms. Consequently, our Prolog-based validation mechanism is capable of reasoning about combinations of different access control models, not only of a given access control model.

The work of Alghathbar incorporates RBAC requirements into UML models and validates such requirements by using logic programming [14], which is closer to our work. In comparison, our approach is able to provide support for modeling DAC or MAC or RBAC requirements, or a combination of such access control paradigms. Furthermore, our textual language and Prolog-based verification tool are able to express and reason about a variety of access control models both individually and combined.

6 Conclusions

While there is significant literature on the many recognized access control methods and their extensions, more generic methods that combine the power of established access control models have been less studied. This is probably due to the fact that the combination of methods opens new issues, such as creating models for reasoning about them and multiplying the possibilities of inconsistency, which can be difficult to recognize.

Our language CatBAC starts from a graphical UML-based representation, called UACML, which is flexible enough to represent many data access relationships that can occur in organizations. UACML features the concept of category that can be instantiated to represent hierarchical groups or roles to which subjects can be assigned. CatBAC is constructed on the basis of UACML, and so it can express all the data access relationships that the graphical representation can express. It has an intuitive syntax, so that it is easy to read and write. It is strongly typed, thus reducing the possibility of errors. We mapped the CatBAC textual language to Prolog and demonstrated the usage of Prolog inference engine to find paths of inference through graphs of concepts. This capability shows that CatBAC is executable and is capable of detecting inconsistencies.

Although our initial motivation was to create a methodology for hybrid access control models, the work presented in this paper could be described as a "generic" framework for designing access control models, which subsumes the most important classical models. A detailed justification of this last point will be considered as part of our future work. Future work will also have to dwell on a thorough comparison between CatBAC and the access control models it subsumes. We will also further explore the practical application of CatBAC and its Prolog implementation in realistic organizational applications. Clearly, the method we propose can be the basis of a lifecycle methodology for creating access control systems: from requirements to UACML to CatBAC to verification to implementation. This methodology could only be alluded to in this paper.

Acknowledgment

This work was funded in part by grants of CA Technologies and the Natural Sciences and Engineering Research Council of Canada. We thank Nadera Slimani for her contribution to the previous work on UACML.

References

[1] N. Slimani, H. Khambhammettu, K. Adi, and L. Logrippo, "UACML: Unified access control modeling language," in *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS'11)*, 2011, pp. 1 –8.

[2] S. Barker, "The next 700 access control models or a unifying meta-model?" in *Proceedings of 14th ACM Symposium on Access Control Models and Technologies (SACMAT'09)*, 2009, pp. 187–196.

[3] G.-J. Ahn, "The RCL 2000 language for specifying role-based authorization constraints," Ph.D. dissertation, George Mason University, USA, 1999.

[4] M. G. K. Sohr, G-J. Ahn and L. Migge, "Specification and validation of authorisation constraints using UML and OCL," in *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, ser. LNCS, vol. 3679, 2005, pp. 64–79.

[5] G.-J. Ahn and M. E. Shin, "Role-based authorization constraints specification using object constraint language," in *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'01)*, 2001, pp. 157–162.

[6] I. Ray, N. Li, R. France, and D.-K. Kim, "Using uml to visualize role-based access control constraints," in *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT'04)*, 2004, pp. 115–124.

[7] D. W. L.Wang and S. Jajodia, "A logic-based framework for attribute based access control," in *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, 2004, pp. 45–55.

[8] P. Epstein and R. Sandhu, "Towards a UML based approach to role engineering," in *Proceedings of 4th ACM workshop on Role-based Access Control (RBAC'99)*, 1999, pp. 135–143.

[9] M. Shin and G. Ahn, "UML-based representation of role-based access control," in *Proceedings of 9th IEEE International Workshops on Enabling Technologies (WETICE'00)*, 2000, pp. 195–200.

[10] T. Doan, S. Demurjian, T. Ting, and A. Ketterl, "MAC and UML for secure software design," in *Proceedings of 2004 ACM workshop on Formal methods in security engineering (FMSE'04)*, 2004, pp. 75–85.

[11] D. Basin, J. Doser, and T. Lodderstedt, “Model driven security: From UML models to access control infrastructures,” *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 39–91, 2006.

[12] J. Pavlich-Mariscal, S. Demurjian, and L. Michel, “A framework of composable access control features: Preserving separation of access control concerns from models to code,” *Computers & Security*, vol. 29, no. 3, pp. 350–379, 2010.

[13] I. Ray, N. Li, D. Kim, and R. France, “Using parameterized UML to specify and compose access control models,” in *Proceedings of 6th IFIP WG 11.5 Conference on Integrity and Control in Information Systems (IICIS)*, 2003.

[14] K. Alghathbar, “Validating the enforcement of access control policies and separation of duty principle in requirement engineering,” *Information and Software Technology*, vol. 49, no. 2, pp. 142–157, 2007.