A Generic Attribute-Improved RBAC Model by Using Context-Aware Reasoning

Chen-Chieh Feng and Liang Yu
Department of Geography, National University of Singapore, Singapore

Abstract - *Traditional role-based access control models (RBAC) are restricted in certain domains and difficult to extend for other applications. The problem is exacerbated with the need to use different RBAC models to manage a variety of resources. Rather than changing the entire RBAC, we propose a generic access control model that takes the advantage of RBAC's reasoning capability and then extends it by adding attributes which significantly improve the inference process. An algorithmic framework is depicted with customizable interfaces that adapts to the requirements of the users. The generic access control model potentially can be adapted to a broader range of user requirements.*

Keywords: access control model, RBAC, attribute

1 Introduction

Access control has been recognized as an important feature of data-centric applications where data are required to be shared under a certain policy [1, 2]. In an information system, the typical use case involves a user executing an operation over an object. The access control engine is expected to give a yes or no answer based on a set of rules. Many fundamental models have been proposed among which the role-based access control (RBAC) model [3] is the most popular mainly because it supports reasoning on the *user-role* and *superRole-subRole* relations.

RBAC model has been improved to meet requirements of various information systems. One type of extension explicitly considers spatial, temporal, and spatiotemporal dimensions. Access control models based on spatial attributes, e.g., SRBAC [4] or GEO-RBAC [5], are used in spatial applications. These models restrict a user's roles according to the spatial region in which the user is located and are extremely useful in wireless location based applications [5]. Temporal role-based access control (Temporal-RBAC) model [6] adds temporal restrictions to the user roles and the hierarchical relations between two roles [7]. The restrictions make it possible to dynamically enable or disable relations between users, roles, and rules, and to automatically disable the access to the associated element after the life-cycle is over. Spatial-temporal role-based access control (STRBAC) has also been proposed [8] to restrict the roles of both subject and object according to the spatiotemporal information.

A second type of RBAC extension considers workflow in organizations, such as task role-based access control (Task-RBAC) [9]. In Task-RBAC, a task reflects different job assignment of a workflow and is a finer unit than a role. It forms the middle layer between subject and object. Organization role-based access control (ORBAC) [10] suggests that all the assignments and authorizations have to be associated with organizations. ORBAC is further extended to incorporate a mechanism to handle access control in a distributed environment, which was named O2O [11].

Despite of these achievements, existing access control models still suffer from the following shortcomings in an enterprise application:

1) The authorization is only performed at an abstract level. It only uses abstract elements such as role, activity and view. A concrete element such as a real user cannot be authorized directly. The authorization process is thus inflexible when we need to authorize a privilege to a concrete element.
2) The models described above cannot be easily extended or customized. Every model requires a set of compulsory elements, e.g., ORBAC requires that every role is under one or more organizations while Task-RBAC requires every role is associated with a task. This makes it difficult to consider more elements or remove some elements from a model.
3) Different models are not subject to the same basic model, making communications between different access control systems difficult. One may argue that the O2O model [11] solve this problem. However, O2O requires all the organizations to use the ORBAC model. A user from system A should be defined in a virtual organization A-B to gain authorization in system B. Nevertheless one organization always has no knowledge about users from another.

Other than RBAC-based models, recently a more flexible model based on a set attributes that an user could prove to have (e.g., clearance level), termed attribute-based access control (ABAC) [12], has been proposed to either replace RBAC or at least simplifies RBAC and makes it more flexible [13]. Attribute has a simple form so it is easy to be shared by different organizations. A system does not have to authorize a user in advance but authorize by the user's

This work was supported the project *CyberInfrastructure of Center for Environmental Sensing and Modeling @ Singapore-MIT Alliance for Research and Technology* (*CENSAM @ SMART*).

Chen-Chieh Feng is with Department of Geography, National University of Singapore: 1 Arts Link, Singapore 117570. (e-mail: geofcc@nus.edu.sg).

Liang Yu was with Department of Geography, National University of Singapore. (e-mail: liangyu.geo@gmail.com).

attributes instead. However, the trade-off for this flexibility is the complexity of cases that must be considered – negotiation between parties must establish trust using elements' attributes and ensure that parties use the same definition for attributes [13]. ABAC also suffers from the lack of support for role-based reasoning.

In this paper, we present a generic access control model that combines the advantages from RBAC and ABAC. We extend the reasoning ability of RBAC by introducing a generic hierarchical relation rather than specific relations such as *user-role*. The notion of attribute from ABAC is used to form *context* which determines the applicable scope of rules. A standard validation algorithm with customizable interfaces is proposed. The customizable interfaces are important feature of the proposed generic model considering most access control requirements are often discovered after the implementation of a system [14].

The remainder of the paper is organized as follows. Section 2 presents the generic model as well as its fundamental definitions and theorems. Section 3 discusses how to add context to the model and use it to supervise the reasoning process. Section 4 proposes a framework of validation algorithm and demonstrates a prototype implemented with Java, and Section 5 concludes the work and lays out the future work.

2 Generic attribute-based RBAC model

2.1 A simple use case

The simplest access control case can be described as a binary relation between a *subject* and an *object*, where a subject refers to a user and an object refers to various resources. To distinguish one access privilege from another, e.g., read from write, the relation becomes a triple which include one more element, i.e., *operation*. An authorization is then described as a pattern $p = (sub, opt, obj)$. The concept of *role* was introduced to group subjects. By assigning privileges to a role, all subjects assuming the role are automatically granted the privileges which have been assigned to the role. A simple example is as follows:

- *Tom* is an *analyst*
- An *analyst* can *read annualReport.xls*
- Inference: *Tom* can *read annualReport.xls*

The statements above contain three essential element types of RBAC:

- Individuals: *Tom*, *analyst*, *read*, and *annualReport.xls*
- Type assumption relation: $r_1 = <analyst, Tom>$, indicating that Tom is an analyst
- Approved authorization pattern: p_1 = {*subject=analyst*, *operation=read*, *object=annualReport.xls*}

Note that individual elements in p_1 are its attributes and can be denoted as $p_1.subject = analyst$, $p_1.operation = read$, and $p_1.object = annualReport.xlst$.

If *Tom* wants to read the *annualReport.xls*, the validation process needs to decide if pattern p_2 = {*subject=Tom*, *operation=read*, *object=annualReport.xls*} is approved. Given the type assumption relation r_1, the pattern p_2 does not have to be defined directly but asserted by p_1. We note this as $assert(p_1, p_2) = true$. The assertion process can be further utilized for cases in which role hierarchy defined as a sub-role inherits from its super-roles all the features. For example, assume that *seniorAnalyst* is a sub-role of *analyst*, denoted as $r_2 = <analyst, seniorAnalyst>$, and *John* is a *seniorAnalyst*, denoted as $r_3 = <seniorAnalyst, John>$. The request for permitting John to read the *annualReport.xls*, denoted as pattern

- *p3 = {subject = John, operation = read, object = annualReport.xls}*,

can also be approved without specifying explicitly the pattern p_3 but again asserted by p_1 (assert(p_1, p_3) = true) because the existence of both r_2 and r_3.

Note that the three patterns p_1, p_2, and p_3 serve different purposes. The p_1 is an approved pattern for authorization while the p_2 and the p_3 are undecided input patterns for validating requests. The three relations carry different semantics. The relation r_1 and r_3 are *type-assumption* relations between a concrete and an abstract element, while the relation r_2 indicates an inheritance relation between two abstract elements.

The relations stated above stands for a partial ordering, termed *hierarchy*, between the first and the second elements, denoted by '$\succ$' below.

- $r = <e_1, e_2> \equiv e_1 \succ e_2$

Using $r_1 – r_3$ as examples,

- $r_1 = <analyst, Tom> \equiv analyst \succ Tom$
- $r_2 = <analyst, seniorAnalyst> \equiv analyst \succ seniorAnalyst$
- $r_3 = <seniorAnalyst, John> \equiv seniorAnalyst \succ John$

The following axiom follows immediately after r is defined:

- $e_1 \succcurlyeq e_2 \equiv (e_1 = e_2) \cup (e_1 \succ e_2)$

The transitivity axiom of a partial ordering relation can then be used to infer new hierarchical relations. The validation process is to search the approved patterns which assert the input patterns, where the hierarchical relations are used for reasoning. In the above example, the assertion process can be decomposed as:

- $assert(p, p') \equiv (p.subject \succcurlyeq p'.subject) \cap (p.operation = p'.operation) \cap (p.object = p'.object)$

2.2 Improvement and Formalization

The original RBAC model discussed in Section 2.1 has been improved to meet more sophisticated access control requirements. These efforts can be classified into three categories according to their functionalities:

1) Elements to elaborate the hierarchical relations. More element types are entitled to have roles, e.g., operation and object in ORBAC are now associated with *Activity* and *View*.
2) Elements to constrain the domain. For example, in the Temporal-RBAC, a role is only enabled in a temporal region, so does the user-role assignment. In the ORBAC model, a role is enabled within an organization, so does

a superRole-subRole relation.

3) Elements to resolve the conflict. This type of improvement incorporates richer authorization types as well as the solutions for resolving conflicts between them. *Prohibition* and *Obligation* are two exemplar authorization types, in addition to *Permission*, being considered [15]. The type can also be considered as a specific attribute, so the pattern p_3 could be reformatted as

- p_3 = {*type=permit, subject=analyst, operation=read, object=annualReport.xls*}.

The first point indicates that the hierarchical relation can be applied to any entities. With that, new patterns can be inferred by replacing every element with its sub-elements. The assert method can be rephrased as:

- *assert*(*p, p'*) ≡ (*p.type* ⩾ *p'.type*) ∩ (*p.subject* ⩾ *p'.subject*) ∩ (*p.operation* ⩾ *p'.operation*) ∩ (*p.object* ⩾ *p'.object*)

Type can also have hierarchies, e.g., *Obligation* can be seen as a sub-type of *Permit*. Based on the analysis above, the essential elements in all the RBAC-based models are depicted in Figure 1.

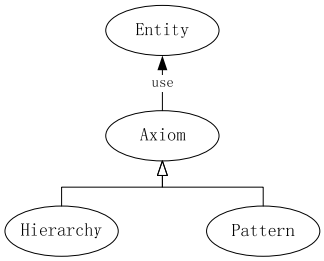

Figure 1. Basic elements in a access control model

Definition 1. A generic role-based access control (*GRBAC*) model is a tuple *GRBAC* = (*ES*, *AS*), where *ES* is a set of entities and *AS* is a set of axioms. Each *AS* is also a tuple *AS* = (*HS*, *PS*), where *HS* is the hierarchy set and *PS* is the approved pattern set.

In the generic model, *entity* represents any individuals since it can be divided into different categories in a concrete model for the management purpose. For example, *user*, *dataset*, *service*, *task*, *organization*, *space*, and *time* are all entities. *HS* is a set of hierarchical relations such as the r_1 and r_2 that mentioned above. *PS* is a set of approved patterns such as p_1.

Definition 2. A hierarchical relation $hr = <e_1, e_2>$ is a partial order relation between two entities, e_1 and e_2, for which e_1 is superior to e_2, denoted as $e_1 > e_2$. The e_1 and the e_2 can be considered as attributes to *hr*, denoted as $hr.super = e_1$, $hr.sub = e_2$.

Two most common hierarchical relations are type assumption between subjects and inheritance of roles. Hierarchical relations can also be established across different entity types. For example, in the hierarchical relation r_4 below, a role (*analyst*) is associated to a task (*analysis*) as its subordinate entities:

- r_4 = <*analysis, analyst*> ≡ *analysis* > *analyst*, if
- p_4 = {*subject= analysis, operation=read, object=annualReport.xls*}

The *assert*(p_4, p_3) is thus evaluated to be true because

- *analysis* > *analyst* > *seniorAnalyst* > *John*

The partial order relation can also be applied to the pattern. The fact that an approved pattern asserts an input pattern means it has a same or broader semantics.

- *assert*(*p, p'*) ≡ *p* ⩾ *p'*

Definition 3: Pattern $p = \{a_1, a_2, \ldots, a_n\}$ is a collection of attributes. Each attribute is a tuple $a_i = <k_i, v_i>$, where k_i is the key and v_i is the value. It can be noted as $p.k_i = v_i$.

Note that both *hierarchy* and *pattern* are expressed by *attribute*, which is a key-value pair <*k, v*>. The key is used to uniquely identity the attribute type and the typical key is an attribute name denoted by a string. The value can be simply an entity or a collection of entities, which means that an attribute can have sub-attributes.

An entity in GRBAC can belong to different entity types. This design choice allows the model to maintain maximum flexibility for defining a pattern. For example, a *service* could be an *object* in a pattern while as a *subject* in another. It also enables the authorization over concrete entities rather than just abstract ones.

Attributes of hierarchical relation and pattern can also be extended. To understand how a pattern is extended, consider the following example with temporal attributes. Assuming two input patterns ip_1 and ip_2 and an approved pattern ap_1 are specified:

- ip_1 = {*type=permit, subject=Tom, operation=read, object=map1, time='13th Jan 2009'*}
- ip_2 = {*type=permit, subject=Tom, operation=read, object=map1, time='13th Jan 2010'*}
- ap_1 = {*type=permit, subject=analyst, operation=read, object=SpatialData, time='2009'*}

Given that *Tom* plays role *analyst* and *map1* is considered as *SpatialData*, ip_1 is asserted by ap_1 while ip_2 is not, because ip_1's lifecycle is within the time period of ap_1 but ip_2's is not. By using the hierarchical relation *'2009'* ⩾ *'13th Jan 2009'* establishes while *'2009'* ⩾ *'13th Jan 2010'* does not. The rule can be described as:

- *ap* ⩾ *ip* ≡ (*ap.type* ⩾ *ip.type*) ∩ (*ap.subject* ⩾ *ip.subject*) ∩ (*ap.operation* ⩾ *ip.operation*) ∩ (*ap.object* ⩾ *ip.object*) ∩ (*ap.time* ⩾ *ip.time*)

As discussed earlier in this section, different access control models can be applied in the same environment and they have to communicate with each other. There is a possibility that an attribute defined in one model does not exist in a second model. For example,

- ap_2 = (*type=permit, subject=analyst, operation=read, object=SpatialData*)

This happens when a user from a Temporal-RBAC system wants to access a regular RBAC system. Since each attribute is used as a restriction, the pattern ap_2 should not be limited in any temporal region. Apparently, *assert*(ap_2, ip_1) and *assert*(ap_2, ip_2) can both be established.

Theorem 1: If $ap = \{a_1, a_2, \dots, a_n\}$, $ap \geqslant ip \equiv \forall a_i = \langle k_i, v_i \rangle$, $(ip.k_i \neq \Phi) \cap (v_i \geqslant ip.k_i)$, where *ap* is an approved pattern, *ip* is an input pattern, *a* is an attribute, *k* is a key, *v* is the value of a key, and Φ is null. The input pattern can have more attributes but every attribute in the authorization pattern should be found in the input pattern. For example,

- ap_3 = {*subject=analyst, operation=read, object=SpatialData, organization=Group1*}

The authorization pattern ap_3 does not assert ip_1 or ip_2 because they both lack the attribute *organization*. In our generic model, a pattern is not required to have minimum number of attributes, but sometimes the pattern is meaningless without sufficient attributes. Using the following five patterns as an example:

1) p = {*type=permit, subject=Administrator*}
2) p = {*type=permit, operation=search*)
3) p = {*type=permit, object=SpatialData*}
4) p = {*subject=analyst, operation=read, object=SpatialData*}
5) p = {*type=permit*}

The pattern 1 can be understood as the administrator is permitted to do everything; pattern 2 means an access request will be permitted as long as the operation is *search*; pattern 3 means the *SpatialData* is publicly open to any access. However, the pattern 4 and 5 are not acceptable because the former lacks the authorization type while the later does not have any other attributes besides a type. Thus, we demand that a pattern must have at least two attributes with one of them indicating the authorization type. The other one must be associated with an entity.

In this section the generic RBAC model has been defined with the minimum elements and a flexible authorization model. The validation is decomposed to a set of reasoning processes. The generic model can be extended to support more elaborated authorization rules, which would be discussed in the next section.

3 Context-aware Reasoning

Attributes of an authorization pattern described in Section 2 are not always independent from each other. Certain attributes may have special status as they affect the applicability of a pattern or a hierarchy, and thus the rest of the attributes. Termed *context* in GRBAC, these attributes demand special treatment because the authorization process is significantly affected by restricting the elements it utilizes. In general, additional attributes require additional reasoning process when comparing two patterns. In addition, an attribute as a context can affect the reasoning process in other ways:

1) A context can change the hierarchies established for the input. Using the following patterns and the hierarchical rule as an example,
   - ap_1 = {*type=permit, subject=analyst, operation=read, object=SpatialData*}
   - ip_1 = {*type=permit, subject=Tom, operation=read, object=SpatialData*}
   - hr_1 = {*super= analyst, sub=Tom, organization=Group2*}
   - ap_2 = {*type=permit, subject=analyst, operation=read, object=SpatialData, organization=Group1*}
   - ip_2 = {*type=permit, subject=Tom, operation=read, object=SpatialData, organization=Group1*}

   For ap_1 *and* ip_1, the hierarchy hr_1 can be established and *assert*(ap_1, ip_1) evaluates to true. However, the addition of context organization = *Group1* in ap_2 and ip_2 causes the assertion *assert*(ap_2, ip_2) fails because hr_1 is defined in another organization (*Group2*). The reasoning process using the transitive relation is also affected because all the antecedent hierarchical relations are required to be applicable to the input.
2) A context can change the entities applicable to the input. For example, a user created with a three-day lifespan will be invalidated after three days. A role associated with one organization means it is not applicable to another.

Below formal definitions of the context and its associated rules are given.

Definition 4: *Context* is a specific condition reflected by a collection of attributes, $ct = \{a_1, a_2, \dots, a_n\}$.

Similar to *pattern*, the attributes of a context are also referred to by their keys. In a validation process, the reasoning process is controlled by the context attributes, which means 1) context attributes have a higher priority and 2) context attributes affect the applicability of common attributes.

The validation process is to compare the context from approved pattern and the context from the input pattern to see if the latter is contained by the former one. A context is attached to an axiom as an attribute. For example,

- ip_1 = {*type=permit, subject=analyst, operation=read, object=SpatialData, context=*{*organization=Group2*}}

The *organization* attribute has been moved to the context, which means that this attribute will affect the reasoning through other attributes. The *context* attribute is not compulsory and an axiom without a context means it can be applied in any contexts. The partial operator '$\geqslant$' for hierarchical relation also applies to context. To assert that one context is superior to another, we need to compare each of its attributes are superior to those of another.

Theorem 2: If $ct_1 = \{a_1, a_2, \dots, a_n\}$, $ct_1 \geqslant ct_2 \equiv \forall a_i = \langle k_i, v_i \rangle$, $(ct_2.k_i \neq \Phi) \cap (v_i \geqslant ct_2.k_i)$.

A context can have sub-contexts. The context affects the way of reasoning on each single attribute. A hierarchical relation is used for reasoning only if it is applicable in a specific context. Here the expression $e_1 \geqslant e_2[ct]$ denotes a hierarchical relation between elements e_1 and e_2 is established under a specific context *ct*.

Theorem 3: $e_1 \geqslant e_2[ct] \equiv \exists hr = (e_1, e_2) \in HS$, $(hr.context \geqslant ct)$. It means a conditional partial order relation.

Theorem 4: If $e_1 \geqslant e_2[ct]$ and $e_2 \geqslant e_3[ct]$, then $e_1 \geqslant e_3[ct]$.

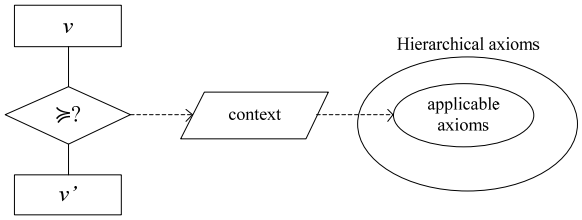

Figure 2. Use context to assert if the hierarchical relation between two attribute values (*v* and *v'*).

As shown in Figure 2, the context defines a subset of the whole axiom set that becomes the axiom set applicable for reasoning. Theorem 4 implies that the context-aware hierarchical relation is a transitive relation. With theorem 3 and 4, new hierarchical relations under context *ct* can be inferred. In Section II we standardize the validation without considering the context based reasoning. Here the validation process is improved as follows:

Theorem 5: If $ap = \{a_1, a_2, \dots, a_n\}$, $assert(ap, ip) \equiv \forall a_i = <k_i, v_i>$, $(ip.k_i \neq \Phi) \cap (v_i \succcurlyeq ip.k_i[ip.context])$.

We use the following example to demonstrate the decomposed process. The example contains two patterns (i.e., ap_1 and ip_1) that have organization, space, and time as the attributes and several hierarchical relations between authorization type (hr_1), role (hr_2), operation (hr_3), resource (hr_4), and organization (hr_5). All hierarchical relations except hr_1 have a time attribute to indicate their life time.

- ap_1 = {*type=obligation, subject=analyst, operation=access, object=SpatialData, context={organization=Group1,time=*T_1 }}
- ip_1 = {*type=permit, subject=Tom, operation=read, object=map1, context= {organization=Group2, time=*T_2}}
- hr_1 = {*super= obligation, sub= permit*}
- hr_2 = {*super= analyst, sub= Tom, context={time =*T_3}}
- hr_3 = {*super= access, sub= read, context={time =*T_4}}
- hr_4 = {*super= SpatialData, sub= map1, context = {time =*T_5}}
- hr_5 = {*super= Group1, sub= Group2, context = {time =*T_6}}

The assertion $assert(ap_1, ip_1)$, according to Theorem 5, can be initially decomposed into five sub-processes:

- $assert(ap_1, ip_1) \equiv$ (1) $(obligation \succcurlyeq permit[ct_{ip1}]) \cap$ (2) $(analyst \succcurlyeq Tom\ [ct_{ip1}]) \cap$ (3) $(access \succcurlyeq read\ [ct_{ip1}]) \cap$ (4) $(SpatialData \succcurlyeq map1\ [ct_{ip1}]) \cap$ (5) $(ct_{ap1} \succcurlyeq ct_{ip1}\ [ct_{ip1}])$

(1) The hierarchical relations between authorization types are global and not subject to any context. Thus, sub-process 1 always returns *true*.

(2) The hierarchy of role assignment is subject to a context time. To make it less complicated, their hierarchies are explicitly defined as hr_2, hr_3, and hr_4. Thus the sub-processes 2, 3, and 4 can be replaced by asserting the hierarchical relations between their contexts, i.e.,

- (6) $(ct_{hr2} \succcurlyeq ct_{ip1}\ [ct_{ip1}]) \cap$ (7) $(ct_{hr3} \succcurlyeq ct_{ip1}\ [ct_{ip1}]) \cap$ (8) $(ct_{hr4} \succcurlyeq ct_{ip1}\ [ct_{ip1}])$.

(3) Because that the role assignment is only subject to the attribute time, the sub-processes 6, 7 and 8 can be replaced by asserting the hierarchical relations between the time expressions, i.e.,

- (9) $(T_3 \succcurlyeq T_2) \cap$ (10) $(T_4 \succcurlyeq T_2) \cap$ (11) $(T_5 \succcurlyeq T_2)$.

The comparison of two time values is not related to any context, thus the context expressions $[ct_{ip1}]$ for sub-processes (9), (10), and (11) are removed. Note that the example is a trivial case as the context values (time) become the attributes for comparison. In other cases, a context may have sub-context as stated in Theorem 2. An example is shown in the sub-process (12) below.

(4) According to the Theorem 2, the sub-process 5 can be decomposed as

- (12) $(Group1 \succcurlyeq Group2\ [ct_{ip1}]) \cap$ (13) $(T_1 \succcurlyeq T_2)$.

Given the hierarchical relation hr_4, the sub-process (12) can be replaced by

- (14) $(T_6 \succcurlyeq T_2)$

Again, the context restriction in sub-process 13 and 14 are removed because they are not related to any context. Finally, the assertion process can be rephrased as a series of comparison between time expressions.

- $assert(ap_1, ip_1) \equiv (T_3 \succcurlyeq T_2) \cap (T_4 \succcurlyeq T_2) \cap (T_5 \succcurlyeq T_2) \cap (T_1 \succcurlyeq T_2) \cap (T_6 \succcurlyeq T_2)$

In existing research the time-related computation has been well investigated. They are thus not discussed in this paper. Readers interested in these computations are referred to works in [6-8]. This use case does not employ complicated rules such as restrictions of time, space, organization. It also eliminates the complexity of reasoning process for those hierarchical relations that have not been explicitly defined but can be inferred. It can be implemented according to theorem 4, which would be demonstrated in the next section.

4 Validation algorithm and prototype

Based on the rules defined in the last section, this section demonstrates how GRBAC can be implemented and used in a practical environment. The demonstration focuses on searching the authorization patterns that assert an input pattern. The execution workflow is depicted in Figure 3.

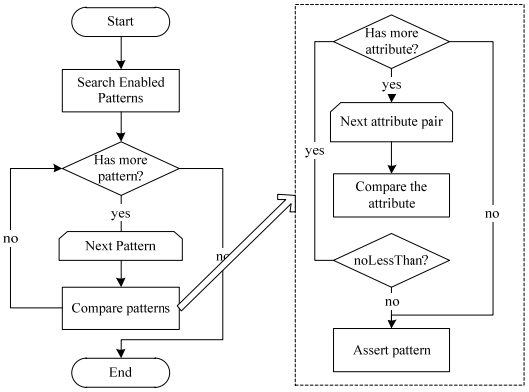

Figure 3. Execution Workflow for Validation

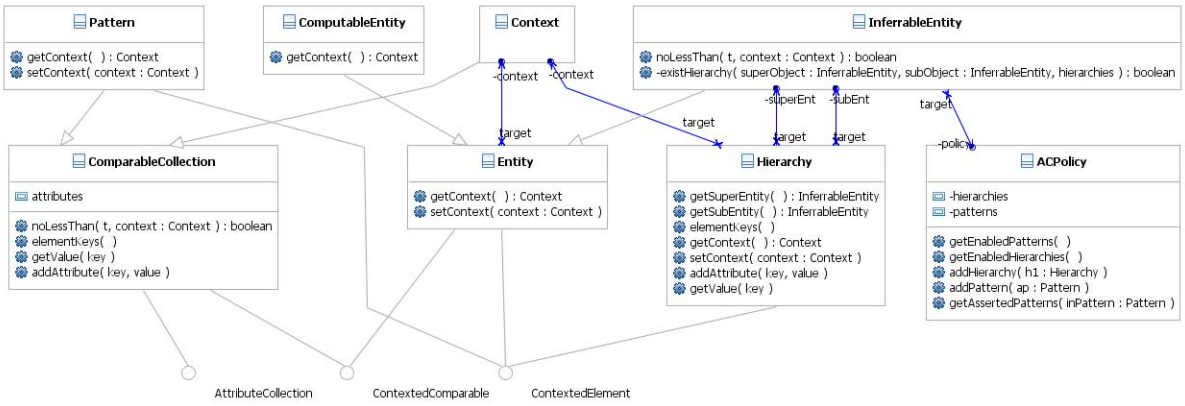

Figure 4. Class Diagram of the Prototype

The algorithm can be roughly divided into three steps:

1) Identify all the enabled authorization patterns.
2) For each candidate authorization pattern, decide if it asserts the input pattern by comparing each pair of attributes with the same key.
3) Return all the passed authorization patterns as the result.

Step 2 plays a crucial role in authorizing resources and thus warrants further explanation specifically on how the search for existing hierarchical relations and the inference of hierarchical relations through reasoning are completed.

In the prototype, the second step is implemented in a way that the whole process can be decomposed into more dedicated units, which in turns are open for customization and then change the behaviours of the access control system.

Figure 4 shows the UML diagram of the core classes in our prototype which is implemented with Java. Several new elements are introduced in the prototype systems in addition to those defined in Section 2 and 3:

1) `ACPolicy` is introduced as a rule system for access control, which consists of *hierarchies* and *patterns*.
2) Three interfaces and one abstract class are introduced to handle attributes. Interface `ContextedElement` indicates that an element is subject to a context object; Interface `ContextedComparable` indicates an object is comparable with another under a certain context; Interface `AttributeCollection` is implemented by Context and Pattern which means an object can be treated as a *key-value* collection; Abstract class `ComparableCollection` implements the algorithm of comparing two collections.
3) The *Entity* class is extended by two sub-classes `InferrableEntity` and `ComputableEntity`. The sub-class *InferrableEntity* implements a comparing method (`noLessThan`) that compares two entities based on a set of hierarchical relations. It takes the hierarchical rules (*hierarchies*) in an `ACPolicy` as input so that its instances only need to customize contexts. The `ComputableEntity` refers to the entities which can be compared by algorithms. These entities usually do not have any context.

```
Class: ACPolicy
   public Collection<Pattern> getAssertedPatterns(Pattern inPattern) {
      Collection<Pattern> patterns = this.getEnabledPatterns();
      Collection<Pattern> result = new ArrayList<Pattern>();
      for (Pattern pattern : patterns) {
         if (pattern.noLessThan(inPattern, inPattern.getContext())) {
            result.add(pattern);
         }
      }
      return result;
   }
Class: ComparableCollection
     public boolean noLessThan(Object another, Context context) {
      Set<Object> keys = this.elementKeys();
      if (another instanceof AttributeCollection) {
         boolean noLessThan = true;
         AttributeCollection attrCollection = (AttributeCollection)
another;
         for (Object key : keys) {
            ContextedComparable thisValue = (ContextedComparable) this
                  .getValue(key);
            ContextedComparable anotherValue = (ContextedComparable)
attrCollection.getValue(key);
            if (!thisValue.noLessThan(anotherValue, context)) {
               noLessThan = false;
               break;
            }
         }
         return noLessThan;
      }
      return false;
   }
Class: InferrableEntity
   public boolean noLessThan(Object t, Context context) {
      Entity entity = (Entity) t;
      if (this.equals(entity)) {
         return true;
      }
      Collection<Hierarchy> hierarchies =
this.policy.getEnabledHierarchies();
      Collection<Hierarchy> applicableHierarchies = new
ArrayList<Hierarchy>();
      for (Hierarchy hierarchy : hierarchies) {
         if (hierarchy.getContext().noLessThan(context, context)) {
            applicableHierarchies.add(hierarchy);
         }
      }
      return existHierarchy(this, (InferrableEntity) entity ,
hierarchies);
   }

     private boolean existHierarchy(InferrableEntity superObject,
     InferrableEntity subObject, Collection<Hierarchy> hierarchies) {
      for (Hierarchy hierarchy : hierarchies) {
         if (hierarchy.getSuperEntity().equals(superObject)) {
            if (hierarchy.getSubEntity().equals(subObject)) {
               return true;
            } else if (existHierarchy(hierarchy.getSubEntity(),
subObject,
                  hierarchies)) {
               return true;
            }
         }
      }return false;
   }
```

Figure 5. Methods for the Validation

Figure 5 list three important methods for the validation. The validation starts from the `getAssertedPatterns` method of `ACPolicy`. It gets all the enabled patterns in the policy and iteratively asserts if each of them assumes a `noLessThan` relation with the input pattern under its context.

The `noLessThan` is an abstract method defined in interface `ContextedComparable`. It has two implementations. The first is in the class `ComparableCollection`, where it is decomposed into iterative calls to all the `noLessThan` methods of its attribute values and compared with the values of the same attribute keys from another collection. The second implementation is in the class `InferrableEntity`, where it uses the hierarchical relations defined in the policy to decide if an entity is in a superior position comparing to another one. The `existHierarchy` method performs the reasoning on the hierarchical relations to infer both direct and indirect hierarchical relations between two entities.

Users can customize the classes to handle more elaborate entities to meet their systematic requirements. The `noLessThan` method can be overridden to support more complicated comaprison algorithm. The most convenient way is to make them sub-classes of either `InferrableEntity` or `ComputableEntity` and reuse the algorithms. The rules can be customized by adding customized context or re-implementing the `noLessThan` method. Entities such as *Role*, *View*, *Dataset*, *Organization*, should extend the `InferrableEntity` and customized the contexts for their instances. The hierarchical relations between entities should also be defined. For custom entities, such as customized temporal or spatial representations, the `ComputableEntity` should be extended and the `noLessThan` method should be customized. For example, for the use case we discussed in the last section, a simple temporal entity can be created as a subclass of `ComputableEntity` which contains the starting and ending date. The algorithm logic used in `noLessThan` method could be:

- $T_1 \geqslant T_2 \equiv T_2$.startingDate $\geqslant T_1$.startingDate $\cap$ T_1.endingDate $\geqslant T_2$.endingDate

Similarly, spatial algorithms can be reused for spatial-related authorization, e.g., to create a geometry class and reuse the algorithm to assert if a spatial region contains another one.

5 Conclusion and future work

We have developed a generic RBAC (GRBAC) model based on context-based reasoning. The traditional role-based architecture is replaced by more generic *hierarchy* and the authorization expression is no longer subject to a fixed sequence. A partial order relation pins the foundation of reasoning and is applied to all elements. The reasoning process is supervised by *context* which is composed of a set of attributes. A prototype has been implemented according to a couple of theorems which can be reused to accommodate more attributes to solve domain-specific access control problems. Various access control functions can be integrated under this framework. Compared to the former works, GRBAC focuses on the flexibility rather than specific domain models. Yet, more remain to be done to adapt it well to a real information system especially in a distributed environment, such as the exchange of authorization information within a distributed environment, the development of advanced rule editing tools, and investigating of the efficiency problem.

6 References

[1] Barth, A., et al., *Privacy and utility in business processes*. 20th IEEE Computer Security Foundations Symposium. 2007. 279-291.

[2] Basin, D. *Model driven security*. in *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*. 2006.

[3] Ferraiolo, D.E., J.A. Cugini, and D.R. Kuhn. *Role-based access control (RBAC): features and motivations*. in *Proceedings of 11th Annual Computer Security Applications Conference, 11-15 Dec. 1995*. 1995. Los Alamitos, CA, USA: IEEE Comput. Soc. Press.

[4] Hansen, F., V. Oleshchuk, and Ieee, *Spatial role-based access control model for wireless networks*, in *2003 Ieee 58th Vehicular Technology Conference, Vols1-5, Proceedings*. 2004. p. 2093-2097.

[5] Damiani, M.L., et al., *GEO-RBAC: A spatially aware RBAC.* Acm Transactions on Information and System Security, 2007. **10**(1).

[6] Bertino, E., P.A. Bonatti, and E. Ferrari, *TRBAC: a temporal role-based access control model.* ACM Transactions on Information and Systems Security, 2001. **4**(Copyright 2002, IEE): p. 191-223.

[7] Joshi, J.B.D., et al., *A generalized temporal role-based access control model.* Ieee Transactions on Knowledge and Data Engineering, 2005. **17**(1): p. 4-23.

[8] Ray, I. and M. Toahchoodee, *A spatio-temporal role-based access control model*, in *Data and Applications Security XXI, Proceedings*, S. Barker and G.J. Ahn, Editors. 2007, Springer-Verlag Berlin: Berlin. p. 211-226.

[9] Oh, S. and S. Park, *Task-role-based access control model.* Information Systems, 2003. **28**(6): p. 533-562.

[10] El Kalam, A.A., et al., *Organization based access control*. Ieee 4th International Workshop on Policies for Distributed Systems and Networks, Proceedings. 2003. 120-131.

[11] Cuppens, F., N. Cuppens-Boulahia, and C. Coma, *O2O: Virtual Private Organizations to manage security policy interoperability*, in *Information Systems Security, Proceedings*, A. Bagchi and V. Atluri, Editors. 2006. p. 101-115.

[12] Karp, A., H. Haury, and M. Davis, *From ABAC to ZBAC: The Evolution of Access Control Models*. Proceedings of the 5th International Conference on Information Warfare and Security, ed. E.L. Armistead. 2010, Nr Reading: Academic Conferences Ltd. 202-211.

[13] Kuhn, D.R., E.J. Coyne, and T.R. Weil, *Adding Attributes to Role-Based Access Control.* Computer, 2010. **43**(6): p. 79-81.

[14] Devanbu, P.T. and S. Stubblebine, *Software engineering for security: a roadmap*, in *Proceedings of the Conference on The Future of Software Engineering*. 2000, ACM: Limerick, Ireland. p. 227-239.

[15] Cuppens, F., N. Cuppens-Boulahia, and M.B. Ghorbel, *High Level Conflict Management Strategies in Advanced Access Control Models.* Electronic Notes in Theoretical Computer Science, 2007. **186**(Compendex): p. 3-26.