Automated Decomposition of Access Control Policies

Linying Su[1], David W Chadwick[1], Andrew Basden[2], James A Cunningham[2]
[1]*Computing Laboratory, University of Kent, UK,*
[2]*Information Systems Institute, University of Salford, UK*
[L.Su-97][d.w.chadwick]@kent.ac.uk, [A.Basden][J.A.Cunningham]@salford.ac.uk

Abstract

Modern dynamic distributed information systems need access control policies to address controlling access to multiple resources that are distributed. The resources may be considered as a single abstract hierarchical resource. An access control policy at a high level should be able to define who is allowed to use the resources. At lower levels, the policy will address controlling access to concrete resources. By modelling the resource hierarchy, it is possible that low level policies can be automatically produced from the high level policy. These low level policies can then be distributed to the concrete resources that use an existing policy based access control decision system so that the high level policy can be enforced throughout the system. In this paper a model for representing and refining high level policies is presented. Other relevant issues and examples for demonstrating the capability of the policy decomposition (refinement) process are also presented.

1. Introduction

Some, but not all, of the security requirements of Internet applications can be addressed via access control policies [1]. Access control policies are concerned with the definition of access control rules i.e. who is allowed to do what to which resources. This paper presents an approach to defining access control policies for use by multiple resources in a dynamic distributed environment. Such policies are different from resource specific policies because they have to deal with multiple resources that may conceptually be regarded as a whole.

Policy based access control systems can be built today for distributed applications by using a centralised policy decision point (PDP) with a common policy that is used by each resource (see Figure 1). Such a system is available today for Grid applications using Globus Toolkit v3.3 onwards, the GGF SAML Authorisation specification [8] and the PERMIS authorisation infrastructure [2]. This sort of access control infrastructure allows co-ordination between the multiple resources that are being accessed. For example, it would be possible to stop a user from accessing more than 3 GB of store throughout the distributed application. The disadvantage of this configuration is that it is a bottleneck to performance because every request needs to be diverted to the central PDP.

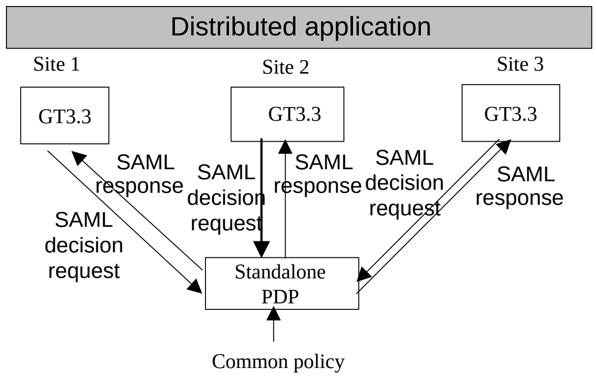

Figure 1. Common policy for today's distributed applications

Alternatively, each site can have its own PDP and the common policy can be distributed to each site based PDP. This approach will increase the performance of the access control decision making, but it lacks co-ordination throughout the distributed application. We envisage that tomorrow's access control systems will be programmable and will distribute copies of resource specific policies to each resource, whilst the PDPs will be linked together to co-ordinate their decision making. This will increase the performance of the decision making and will allow co-ordination between the local PDPs (see Figure 2). A common high level policy for the application can be decomposed (refined) into site specific policies, which are then distributed to each site and only contain policy information relevant for controlling access to that site. In this case, the new site specific policies will become much simpler than today's common policy. In this infrastructure co-ordination between the different decision making systems is needed, although this aspect is not addressed in this paper. We concentrate here solely on the policy decomposition process.

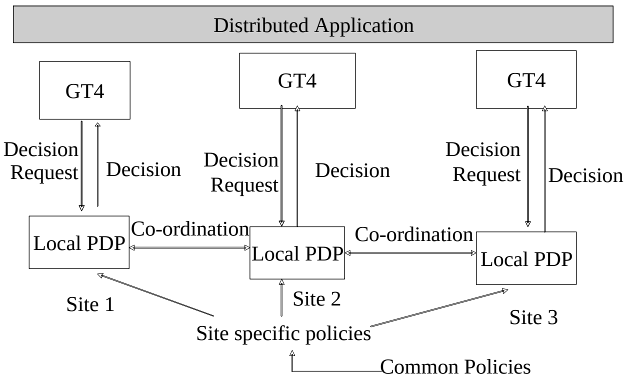

Figure 2. Tomorrow's access control for distributed applications

In this paper, we introduce the concepts of abstract resource types, concrete resource types, resource type hierarchies, resource instances, policy hierarchies and policy decomposition rules.

The multiple resources of a distributed application can be seen as having a hierarchical structure. Policies are needed to control access at each of the levels. At the highest level, the policy is concerned about controlling access to a single abstract resource, for example a particular Grid application. At the lowest level, the policy addresses controlling access to specific concrete resources, such as servers and file stores which make up the abstract resource. Intermediate levels are possible, for example, a computer cluster or a distributed database. We would like to show that policies at any level for whatever abstract or concrete resources can be automatically produced from the high level policy based on policy decomposition rules and resource type hierarchies.

Resource type hierarchies describe how the high level abstract resources are constructed from their lower level concrete and abstract resources. Resource type hierarchies also say what actions (or methods) each of the resources support. A resource instance is an instance of a resource type hierarchy. Multiple instances of the same type hierarchy can occur.

Policy decomposition rules define how high level policies transform into their low level ones so that the policy decomposition (refinement) is realized based on these rules and a resource type hierarchy.

By means of policy decomposition, see Figure 3, existing access control decision systems such as PERMIS and Akenti [3] are able to be applied to multiple distributed resource applications.

Based on an access control policy *P* for an abstract resource *R*, which is a distributed resource, a group of low level policies p_1-p_6 for each component of R can be produced. To do so, we need to establish relationships between the resources. In the example shown in Figure 3, the resource *R* contains six sub-resources, which may be decomposed further into other sub-resources (not shown). This process will be carried out recursively until all sub-resources become site specific concrete resources, where the low level policies are then able to be used to control access to them. This resource decomposition and low level policy production defines a simple policy refinement process.

"Policy refinement is the process of transforming a high-level, abstract policy specification into a low-level, concrete one." [4]. Here we propose this policy refinement process should be one that is able to produce a policy for any resource at any level and ensures that each stage of the decomposition is correct and consistent.

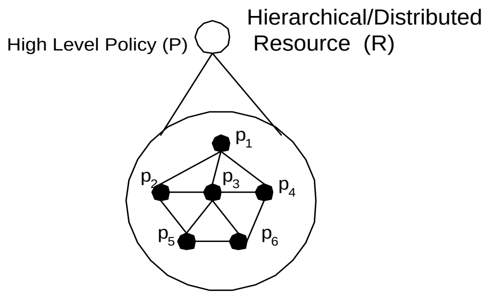

Figure 3. Producing low level policies P_1-P_6 from a high level policy P

In order to achieve this policy refinement, languages that can describe high level policy specifications, resource type hierarchies, and decomposition rules are required. These can then be fed into an inference engine for the low level policies to be inferred. The policy specifications consist of policy components, which describe subjects, resources, actions and conditions. In the refinement process, the policy specifications and resource instances are regarded as facts; the decomposition rules and resource type hierarchy are inference rules. The inference engine can then infer the low level policies based on the facts.

We have developed such an inference engine in the form of a Java API. This engine uses the OWL language [6] to specify the policies and resource type hierarchies and is independent of any policy-based access control application. For example, we use Protégé [5], an ontology and knowledge-base editor, to create the high level access control policies and resource type hierarchies. The low level policies for concrete resources that are inferred may then be transformed into an application specific language such as XACML [7].

The rest of this paper is structured as follows. In section 2 we describe the resource type hierarchy and resource instances in detail. In Section 3 we present the proposed policy specification model which covers the data structures used in the policy components. Section 4 presents the policy refinement process and the decomposition rules. This describes how to refine the high level policy into a set of low level policies according to the decomposition rules and a given resource type hierarchy and resource instance. Section 5 discusses implementation issues. In section 6 an example of a high level policy specification and its low level policies are given. Finally section 7 presents some conclusions and directions for future work.

2. The Resource Type Hierarchy

We use the Directed And/Or Graph (DAOG) as the main structure to define a resource type hierarchy. This describes the hierarchy of resource types and actions.

A restricted DAOG is denoted by G

$$G=\{I,J,O,E\}$$

where I is a set of terminal nodes whose Out Bound Degree (OBD=0) is zero; J is a set of internal nodes (OBD>0); O is a set of origins of the graph; and E is a set of edges. Each origin is a node denoted as N $(N \in O$ and $O \subseteq I \cup J)$ with zero as its In Bound Degree (IBD=0). If there is an edge from node A to B, where $A \neq B$, then we use $<A, B>$ to represent this edge.

Some restrictions, as follows, are applied to the graph. A DAOG cannot be empty; therefore, it contains at least one node (i.e. O contains a single origin). For any $k \in I \cup J$, k is either an *And* node or an *Or* node exclusively. Interpretation of an *And* or an *Or* node is determined by what the graph is used for. Distinguishing *And* nodes from *Or* nodes is significant only when the graph is used for co-ordination between local PDPs (as in Figure 2). For any $k_1, k_2 \in I \cup J$, if there exists the path P_{k1k2} then the graph can not contain the path P_{k2k1}. This restriction prevents the graph from containing a loop. Any node in the graph is reachable from an origin. A path P_{k1k2} defines a series of edges: $<k_1, t_1>$, $< t_1, t_2>$, …, $< t_{n-1}, t_n>$, $< t_n, k_2>$. This restriction ensures we can establish containing (or contained) relationships in definite steps.

2.1. Application of DAOG to Resource Type Hierarchies

A DAOG can be used to represent a resource type hierarchy. Any set of nodes in a resource type hierarchy may have one or more instances. The relationships between these instances are implied by the corresponding relationships in the resource type hierarchy (see figures 9 and 10 for an example).

For a resource type hierarchy, each internal node in the DAOG denotes an abstract resource type and the nodes in the set O indicate the most abstract types. Each terminal node represents a concrete resource type. The nature of a resource type (being abstract or concrete) may change if the DAOG is modified e.g. if new nodes are added or existing nodes removed. The modification of a DAOG reflects the dynamic changes in a multiple resource system, such as removing and/or adding resources types, or spawning tasks in a Grid job. The edges indicate relations of "has part" or "has alternative" between two resources. If an abstract resource type node is an *And* node, it denotes that the resource comprises all its immediate children i.e. all the edges from that node are "has part". If an abstract resource type is an *Or* node it denotes the resource has alternatives i.e. all the edges are "has alternative" (e.g. separate data stores).

When access control decision making takes places, co-ordination is needed between the resource instances of the descendants of an *Or* node, but not between the descendants of an *And* node. This is to enforce restrictions such as "no more than 2GB of storage may be used" (regardless of which alternative store is used).

2.2. Resource Containment Relationship

The resource type hierarchy in a DAOG defines resource containment relationships between those abstract and concrete resource types. Both the "has part" and "has alternative" relation indicate the containment relationship. Given a DAOG and one of its nodes R, all its contained nodes can be found by traversing the graph from the node R down to the terminal nodes. For example, if there is a path P_{RT} in the graph and $R \neq T$ then T is a contained node of R. If A contains B, we say the type (or action) associated with A is a containing type (or action) of the type (or action) that is associated with B. In other words, the type (or action) associated with B is a contained type (or action) that is associated with A.

Using a graph rather than a tree to represent a hierarchical structure, allows the structure to define resource sharing. For example, as shown in Figure 4, the computing resource instance *R* is distributed to three units (S_1 to S_3), where two of them share the same (remote) printing device *Printer.* Based on the DAOG, a policy, such as “you are allowed to use the distributed computing facility *R*”, can be refined into “you are allowed to use Pcs 1 to 5, the scanner and the printer”.

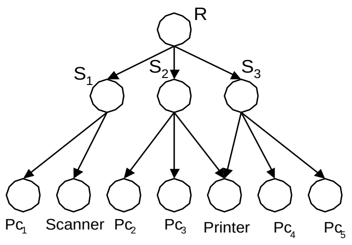

Figure 4. An example DAOG

Regarding the abstract resource instance *R* in Figure 4, it has ten descendants: sites S_1, S_2, S_3, a Printer, a Scanner, and Pc_1 to Pc_5. The node Printer is shared by sites S_2 and S_3. This results in two different paths from R to Printer, which are <R, S_2, Printer> and <R, S_3, Printer>. These two paths indicate that actions on the Printer are implied by actions defined on sites S_2 and S_3. Due to the fact that a node could be shared by different internal nodes, it is possible that different policies for the node could be produced. This may cause a policy conflict. We propose a unified policy refinement approach, which solves the problem of policy conflicts, by creating an access control policy for the shared node that is the conjunctive connection of all its immediate higher level policies. An example is presented in section 5.

2.3. Resource Ontology

Each resource type hierarchy is constructed according to a resource ontology. This ontology provides the rules and vocabulary for creating resource type hierarchies. A resource ontology defines such things as:

- the resource containment rules. These state which abstract resource types or actions can contain which other abstract and concrete resource types or actions.

- the allowable set of attributes that each resource may have.

2.4. Attributes

A resource type hierarchy and a resource instance can be described using a set of attributes. These can be simple attributes, which describe a property of the resource, or complex attributes which describe the hierarchical structure of the resource (i.e. its contained lower level resources) and what actions (along with their parameters) can be performed on the resource. Each attribute comprises 3 components: its name, its value (it may have multiple values) and the data type of the value. E.g. the location of a resource can be represented as a simple attribute with the name “location”, with a value giving the location e.g. “Maxwell building” and the data type is string.

The type and action attributes are mandatory for each resource in a resource type hierarchy. The type attribute has values of data type string. The action attribute has values of data string which comprise the action along with an optional set of parameters (where each parameter is represented as a string). Each non leaf node in a resource type hierarchy must also have the “has part” or the “has alternative” hierarchical structure attribute. Additional attributes may be present in resource instances, but these are ignored if present in the resource type hierarchy.

The hierarchical structure attributes have a much more complex data type, which actually contains embedded attributes. Conceptually these represent the set of child nodes in the DAOG beneath the parent node, and the embedded attributes are the attributes contained in the child nodes. If a child node also has children then the embedded attributes in the parent will themselves be complex attributes with the attributes of the grandchildren embedded in them. In this way the entire DAOG can be represented in the attributes of the origin nodes (an example is presented in section 4.1.2). Note however it is often simpler to represent the DAOG as a set of nodes with simple attributes in each node (as in section 6).

2.5. Action Hierarchy in a Resource Type Hierarchy

A resource type hierarchy not only describes the hierarchical type relationships but also the hierarchical relationships between the actions. For example, the action *drive* on a containing resource of type *car* implies the action *start* on the contained resource of type *engine*, the action *open* on the contained resource *door*, and the action *switchOn* on the contained resource *stereo-system*.

If superior actions associated with a node imply different subordinate actions e.g. *drive* car and *ride in* car would not both imply *start* engine, an appropriate action hierarchy can be achieved in the following way. The resource node that has a multiple value action attribute is split up into a set of new nodes. Each new resource node is given a subset of the action values which imply the same subordinate actions. A further splitting on a subordinate node may also be required when its actions only imply a subset of the actions defined in its subordinate nodes.

Figure 5 gives an example resource type hierarchy, which models both the resource and action hierarchy. In this example the action *drive (taxi)* implies the following actions, *start (engine)*, *open and lock/unlock (door)* and *switchOn/listen (stereo)*. The actions *lock/unlock (door)* imply *lock/unlock (lock)*, whilst the latter imply *use (key)* and *use (remote control)*. However the action *ride (taxi)* only implies *open (door)* and *listen (stereo)*. In figure 5 the nodes in shadow are *And* nodes whilst the clear nodes are *Or* nodes (ignoring the terminal nodes since they do not have any subordinates).

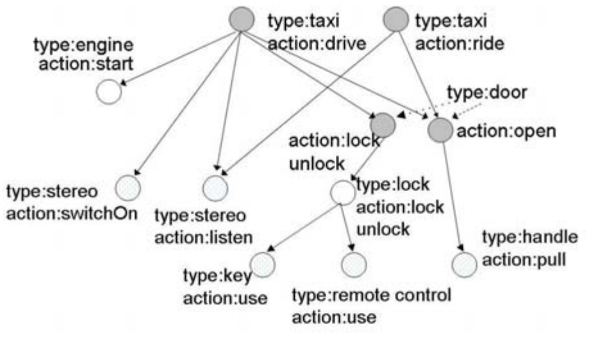

Figure 5. An example DAOG, which also models action hierarchies

3. Policy Specification

An access control policy states *who is allowed to do what actions to which resources under which conditions.* These elements can be specified as a set of filters on all possible subjects, actions, resources, and operational and environmental parameters. If subjects and resources are each described by a set of attributes, then the filters can be used to adopt an Attribute Based Access Control (ABAC) infrastructure [7]. Applying the appropriate filter to the attributes of a subject or resource will find the subject or resource (abstract or concrete) that meets an application's security policy.

An Arithmetic and Logical Expression Tree (ALET) can be used to describe these filters and policy conditions.

3.1. ALET

An ALET is used to model an arithmetic or logical expression, and these are used to describe subject and resource filters. All these filters are then combined into one ALET using AND nodes. A non-leaf node of the ALET tree is either an arithmetic operator {+, -, *, /}, an extended relational operator {=, ≠, >, <, ≥, ≤, ⊃, ⊂, ⊇, ⊆, ∈, ∉} a logical operator {AND, OR, NOT, XOR}, or a function name. An attribute can be considered to be a function where the attribute name is the function name, the attribute value is the function value, the type of the attribute value is the function type, and the function parameter is the object which has the particular attribute. For example, if the location attribute of resource X is Manchester, then the function *location (X)* has the value Manchester.

A leaf node of an ALET could be a constant or a variable. A constant has a type, which can be a string, an integer, a float, a Boolean or a set containing such constants. For example, "student" is a constant with a type of string. A variable has a name, type and value. The value of a variable is set by the user in his access request, or by the underlying system (e.g. time of day). There are four categories of variable: subject variables, resource variables, environment variables and operational (or action parameter) variables, involved in access control policies. A subject variable refers to somebody (e.g. the subject who has role "student") who intends to use a resource. Resource variables represent the resources that are going to be used, for example, the resource that is of type printer. Environment variables are system settings such as time, date, disk quota, etc. Operational variables refer to action parameters set by the user in their requested access e.g. name of file to be opened, or amount of storage requested.

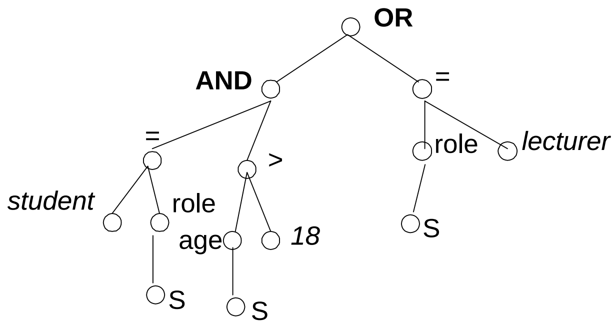

Figure 6. An example ALET

By supporting operational variables in the ALET, we provide a way of addressing the proportion of a resource to be allocated to various users. For example, the relational expression, Memory < 128, states a restriction on the computer memory that can be requested, where Memory is an operational variable.

Figure 6 presents an ALET, which describes "an over 18 student or a lecturer" (i.e. role(S) = student AND age(S) > 18 OR role(S) = lecturer).

3.2. Policy Ontology

Each policy is created according to a policy ontology. This ontology provides the rules and vocabulary for creating access control policies. A policy ontology defines such things as:

- the policy creation rules. These state how an access control policy is created i.e. a policy is an ALET, which has a root.

- the type of an ALET node (e.g. constant, variable, arithmetic operator, logical operator, relational operator and function).

-the relation between the nodes (e.g. an ALET node may have children).

- the attribute of an ALET node (e.g. has integer data type, has function name).

3.3. Policy Component Representation

An access control policy is represented as an ALET, which is a filter to describe subjects or resources of the policy. The resource description includes resource type, actions on the resource, action parameters and other conditions of access. Every policy must contain this component i.e. $action(R)=X\wedge type(R)=Y$, where X denotes a specific action that can be performed on the resource Y. When an access control decision request is sent to a PDP, the policy filters are used to make the decision. If the given subject, action, parameter, resource and environment information can pass the corresponding filter, then this request is granted. Otherwise, it is denied. Literally, the access control policy is a logical expression, which may evaluate to true or false against the given information. If it evaluates to true the request is granted, otherwise the request is denied.

Normally, an access control policy contains parts as subject filters and resource filters. For example, a subject filter is written as an ALET (e.g. $role(S)=student\wedge age(S)>18$) and describes *who* is the subject of the policy. A resource filter written as an ALET is an action, parameter (operational variable), resource and environmental filter, which defines what type of resources are controlled by this policy, what type of actions are allowed on the resources, what type of parameters go along with the actions and what environment settings are required. For example, $(type(R)=car\wedge action(R)=drive) \wedge (\neg(type(R)=stereo \wedge action(R) =switchOn))$ describes a policy for any car resource which can be driven but the stereo cannot be switched on. R is a resource variable.

The default of a subject filter or resource filter is the constant true, which indicates that any subject can access the resource or the subject can perform any action on any resource.

4. Policy Refinement

In order to refine a high level policy into low level policies and then to distribute them to specific sites, we need a process that can manipulate a given overall (high level) policy and transform it into the low level policy for a less abstract or concrete resource type. This new policy then can be further refined for a specific resource instance of the resource type by repeating the refinement process again. Given any node in a DAOG resource instance its policy can be produced in these two steps. It is worth mentioning that if a resource type is subordinate to several superior resources (e.g. as in figure 7) then this could lead to a policy conflict for the subordinate resource. The conflict is resolved by making the policy for this resource type the conjunctive connection of those policies that are produced based on its immediate higher level policies (see section 5 for an example).

The nature of the above policy refinement process ensures that any low level policy for a resource type conforms to its higher level policies. That means, given a specific resource, its higher level policies also evaluate to true when its low level policy evaluates to true.

4.1. Operations Related to Policy Refinement

In this section we define two operations that are required by the refinement process.

4.1.1. Evaluation of relational expression. Given a set of attributes **inf** from a resource (type or instance) and a relational expression **E**, **inf⇒E** denotes an operation that evaluates **E** against **inf**. The relational expression **E** can evaluate to true, false or indeterminate.

inf⇒E can be accomplished in the following two steps:

(1) Substitute function subtrees which are accountable for the evaluation, by the corresponding attribute values from the resource. Functions involved in the expression which are not accountable for the evaluation, are indeterminate.

The accountable functions (i.e. attributes) are the resource type and action attributes when refining policies for a resource type hierarchy, and are all the attributes when refining the policy for a resource instance.

(2) Evaluate the expression. The result of the evaluation can be true or false if all the elements in the expression are determinate. Otherwise the expression returns indeterminate.

When a relational expression evaluates against multiple attribute values, it evaluates to false if none of the value combinations makes it true.

When a relational expression evaluates against an attribute value that is not present in the resource type then the relational expression evaluates to indeterminate.

Here we present examples to show how this calculation is carried out. Given a set of attributes {type:printer, action:print {FileName, Copies}, manufacturer:UK} and relational expressions

(a) action(R)=print

(b) type(R)=scanner

(c) size(R)>1

then (a) evaluates to true because action(R) is replaced with the attribute value print and print=print is true; (b) evaluates to false because type(R) is replaced with printer and printer=scanner is false; (c) evaluates to indeterminate because size(R) is not known at this time.

4.1.2. Replacement of containing type and action. Given a resource type hierarchy, a particular resource node N and a relational expression in the form of action(R)=A or type(R)=T, A or T can be replaced with the action or type attribute value of the resource node N if the resource N is contained by the resource R. This is because a containing action or type implies the contained action or type.

If the action attribute associated with the resource node N has multiple values, then action(R)=A will be replaced with an ALET, which is action(R)=X_1∨action(R)=X_2∨…∨action(R)=X_n. X_i (i=1..n) is one of the action value of the resource N.

Whether the resource R contains the resource N can be determined based on the resource type hierarchy. For example, given a resource type hierarchy for a resource type *car* as a set of attributes,
{type:car, action:drive{}, hasPart:{type:engine, action:start}, hasPart:{type:stereo, action: {switchOn, switchOff}}, hasPart:{type:door, action:open}}
then for a concrete resource type *door* and a policy of action(R)=drive, *drive* can be replaced with *open* because the resource type *car* contains the resource type *door*. Therefore, the action *drive* implies the action *open*.

4.2. Policy Refinement Rules

The policy refinement process, which manipulates a high level policy based on a resource type hierarchy produces the low level policies for any resource type by following the refinement rules. After the policy for a resource type has been produced, these rules may be used again to produce the policy for an instance of this resource type, by using the policy produced for the resource type as the input policy to this (second) refinement process. Note that a policy for a resource instance can never be the policy input into the refinement process.

Given a high level policy *P*, a resource type hierarchy *R* and a particular resource type node *N* in R as input, then in order to get the output, the refined policy $P_{N,}$ for the given node *N*, we apply the following rules in series to the policy $P_{I,}$, where initially P_I= *P* and having applied the rules, P_I will become the refined policy for the node *N* in *R*.

4.2.1. Make all types and actions within the policy specific to the node *N if possible*. For each type or action value $\{a_1, a_2, ..., a_n\}$ that is explicitly mentioned in the policy P_N and the type or action value *{type:*b_1*, action:*b_2*}* associated with the node *N*, if, by traversal of *R* we infer that a type or action a_j (j=1..n) in the policy P_N is a containing type or action of the type b_1 or action b_2 associated with *N*, then we replace a_j in P_N with the contained type or action (b_1 or b_2). In the special case where there are multiple actions associated with the node *N* then we replace each specific relational expression in the form of action(R) = a_j mentioned in P_N with the ALET action(R) = b_{21}∨…action(R) = b_{2m}, where b_2 denotes the action attribute, which contains the multiple values b_{21},…,b_{2m}.

The reason to do this when producing a refined policy P_N for the node *N* is that we want the types and actions mentioned in P_N to be specific to the node *N* if possible.

If the policy addresses contained types or actions then we cannot replace them with the containing types or actions from node *N* since we have not yet reached the contained nodes in the hierarchy. The contained type and action must be preserved for evaluation lower down the DAOG.

This rule is not applicable when producing the policy for a resource instance from the policy for its resource type, because the actions are already specific to this node.

4.2.2. For Instance Refinement, evaluate all relational expressions not involving an action or a type. For each relational expression *f(x, y)* (e.g. x = y) not involving an action or a type in the policy P_N we determine if the expression evaluates to true, false or indeterminate. If the expression *f(x, y)* evaluates to true or false then we replace that expression in P_N with the corresponding value TRUE or FALSE.

Any relational expressions which do not indicate an action or type can be replaced with a TRUE/FALSE value since they will always evaluate to that value under all run time circumstances (e.g. location(R)=UK).

A relational expression that evaluates to indeterminate must be preserved because during refinement we do not know what value it will take at run time.

4.2.3. Evaluate all relational expressions involving an action or type. For each relational expression *f(x, y)* that involves an action or type in the policy P_N we determine if the expression evaluates to true, false or indeterminate. If f(x, y) evaluates to false then we replace that expression with the corresponding value FALSE. The evaluation of contained actions or types in the policy P_N is indeterminate because we do not know any information about the contained actions or types at this time.

Replacing an action or type relational expression that evaluates to false with FALSE is due to the fact that the relational expression need never be evaluated at run time.

If the expression evaluates to true we leave the expression in the policy because it will be needed for run time evaluation by the PDP.

4.2.4. Simplify the policy P_N to produce the refined policy for the node *N*. We now simplify the policy P_N according to the following simplification rules. These rules may be applied iteratively until no more changes take place.

The simplification rules address how Boolean constants can be removed from the ALET, how a tautology or contradiction is identified, and how redundant expressions can be removed from the ALET. By transforming a logical expression into a Disjunctive Normal Form (DNF), these simplifications can be achieved easily. If any term in the DNF contains operational variables, which are not supported by any action in the term, then the term is removed from the DNF. This is because the term always evaluates to false under all circumstances.

The policy P_N is now the refined policy for the type node *N*. The policy for a concrete resource instance can then be produced from its resource type policy by following the above refinement process a second time.

5. Implementation

We have developed a java API, which defines all the necessary classes and methods for creating high level policies and resource type hierarchies and producing various low level policies. This API can be used independently for developing policy-based systems. Users are free to develop their own policy editors for inputting high level policies or to develop backend compilers for translating the produced low level policies into policies written in their existing access control policy languages.

The low level policy for a concrete resources type can be produced by applying the high level policy to the specific resource type straightway. Alternatively, it can be obtained recursively by successive refinement of the policy down the resource type hierarchy. The following example presents the two ways.

Given the resource type hierarchy in Figure 7 and the high level policy as action(R)=use∧type(R)=a (i.e. allowed to use resource of type a), if we apply the policy to the type nodes A, B, C and D, then we can get the following new policies

action(R)=use∧type(R)=a,
action(R)=read∧type(R)=b,
action(R)=write∧type(R)=c,
action(R)=seek∧type(R)=d,

respectively. The same policy for D can also be obtained via B and C by using that nodes policy as the high level policy.

If we have a high level policy such as,

action(R)=use∧type(R)=a∧
¬(action(R)=write∧type(R)=c)

(i.e. "allowed to use resource of type a but not write to c"), then when the policy for D is derived straightaway from this high level policy it is FALSE because it is simplified from action(R)=seek∧type(R)=d∧¬(action(R)=seek∧ type(R)=d). If the policy for D is derived recursively via C it is also FALSE because the policy for C is FALSE. However, if the policy for D is derived via B it becomes action(R)=seek∧type(R)=d. However, according to the policy conflict resolution strategy of conjunctive connection we get the final policy for D as the logical expression action(R)=read∧ type(R)=b∧FALSE, which evaluates to FALSE.

.

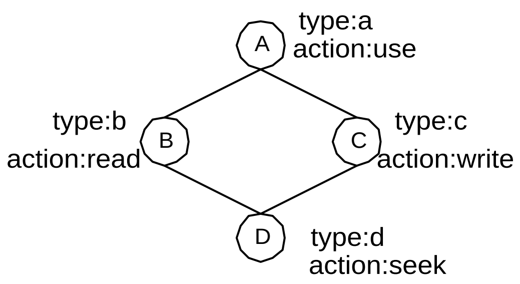

Figure 7. A simple resource type hierarchy

Although the above high level policy specifies the access control for a hierarchical resource type, the policy itself is independent of the resource's hierarchical structure. This means that the above policy could also be applied to other resource hierarchies. For example, if we apply the same policy to another resource hierarchy depicted in Figure 8, we get the policy for node D as action(R)=get∧type(R)=d and the policy for node E as FALSE.

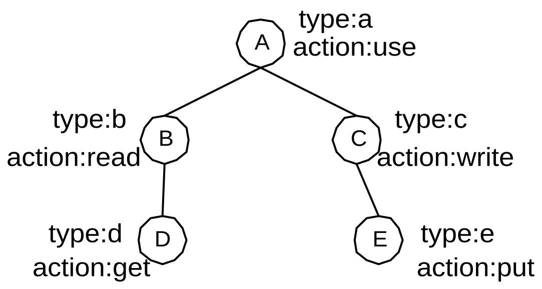

Figure 8. A revised resource type hierarchy

At the moment, we use Protégé as the policy editor, which uses OWL as the policy language to define the high level policies and resource type hierarchies. The OWL language is adequate for this due to the fact that the ALET and DAOG can be implemented straightforwardly in OWL. We use OWL classes to define the various ALET and DAOG nodes, and use OWL properties to describe the relationships between those nodes. The relationship is a parent-child relationship or a having-an-attribute relationship. By creating instances of the OWL classes and linking them through the OWL properties, a particular access control policy or resource type hierarchy can be created.

6. An Example

In this section we demonstrate the ability of our approach to produce a set of low level policies for both resource types and resource instances.

6.1. Resource Type Hierarchy and High Level Policy

Let VO (Figure 9) denote a resource type hierarchy for a *virtual organisation* type resource, which consists of *organisation* type resources which may donate various concrete resources such as printers, filestores, scanners, PCs, and web access to the VO. The VO resource type has one action *use* with no parameters, whilst the organisation resource type has the action *access* with no parameters. The low level concrete resource types, their associated actions and parameters are: Pc(use), Web (browse{URL}) , Printer (print {FileName, Copies}), Scanner (scan{Destination}), and FileStore (write{FileName, Mode, Size}).

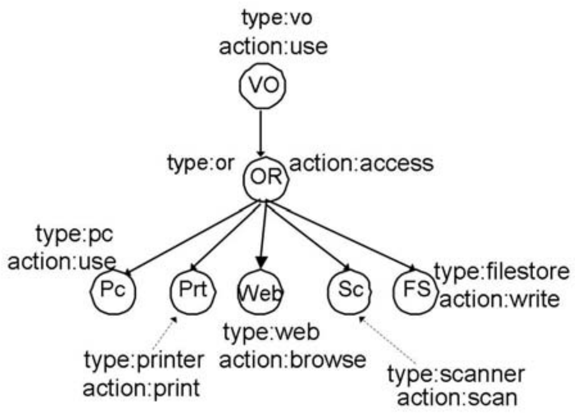

Figure 9. The resource type hierarchy for the VO abstract resource

The complete resource type hierarchy specification for the VO resource is defined in the following attributes:

Att(VO)={action:use, type:vo, hasPart:Att(OR)}.

Att(OR)={action:access, type:or, hasPart:Att(Pc), hasPart:Att(Prt), hasPart:Att(Web), hasPart:Att(Sc), hasPart:Att(FS)}

Att(Pc)={action:use, type:pc}

Att(Prt)={action:print{Filename,Copies}, type:printer }

Att(Web)={action:browse{URL}, type:web}

Att(Sc)={action:scan{Destination}, type:scanner}

Att(FS)={action:write{Filename, Mode, Size}, type:filestore}

Figure 10 represents a VO instance comprising organisations whose owners are the Universities of Oxford, Cambridge and Manchester. Oxford donates a

PC located in the UK and a printer located in France to the VO. Cambridge donates web access and a filestore, both located in the UK, whilst Manchester donates a filestore and a scanner, both located in the UK. Note that the type and action attributes are not shown, but are inherited from the type hierarchy.

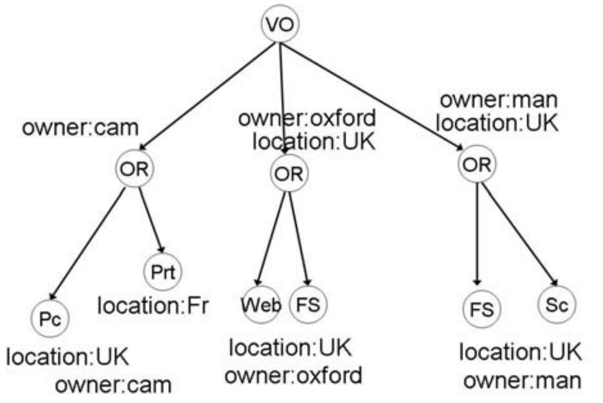

Figure 10. A VO resource type instance

The complete resource instance specification for this VO resource is defined in the following attributes:

Att(VO)={action:use,type:vo, hasPart:Att(OU), hasPart:Att(CU), hasPart:Att(MU)}

Att(CU)={action:access, type:or, owner:cam, hasPart:Att(Pc), hasPart:Att(Prt)}.

Att(OU)={action:access, type:or, owner:oxford, hasPart:Att(Web), hasPart:Att(FS1}

Att(MU)={action:access, type:or, owner:man, location:UK, hasPart:Att(Sc), hasPart:Att(FS2)}

Att(Pc)={action:use, type:pc, location:uk },

Att(Prt)={action:print{Filename, Copies}, type:printer, location:Fr}

Att(Web)={action:browse{URL}, type:web, location:UK }

Att(FS1)={action:write{Filename, Mode, Size}, type:filestore, location:UK }

Att(Sc)={action: scan{Destination}, type:scanner, location:UK }

Att(FS2)={action:write{Filename, Mode, Size}, type:filestore, location:UK }

Given a high level access control policy for the virtual organisation of say "During 9:00-18:00, any student can use any virtual organisation resource located in the UK except they cannot print more than 2 copies on printers or write files greater than 1MB to filestores which belong to Manchester University," this can be represented as,

role(S)=student∧location(R)=UK∧
action(R)=use∧type(R)=vo∧
(¬(action(R)=print∧type(R)=printer∧print{Copies} >2))∧(¬(action(R)=write∧owner(R)=man∧
type(R)=filestore∧write{Size}>1MB))∧
Time≥9:00∧Time≤18:00.

6.2. Policies for Resource Types and Instances

The policy for the VO type remains the same as the initial high level policy because type(R)=vo and action(R)=use evaluate to true; location(R)=UK, owner(R)=man, Copies>2, Size>1MB and Time≥9:00∧Time≤18:00 are all indeterminate, whilst action (R)=print (or write) and type (R)=printer (or filestore) evaluate to indeterminate (according to 4.2.3). The policy for the OR type is

role(S)=student∧location(R)=UK∧
action(R)=access∧type(R)=or∧
(¬(action(R)=print∧type(R)=printer∧print{Copies) >2))∧(¬(action(R)=write∧owner(R)=man∧
type(R)=filestore∧write{Size}>1MB))∧
Time≥9:00∧Time≤18:00.

since, according to 4.2.1 we replace the containing type (vo) and action (use) in the policy with the contained type (or) and action (access).

The policies for the OR instances owned by Oxford and Manchester are refined to become

role(S)=student∧action(R)=access∧
type(R)=or∧Time≥9:00∧Time≤18:00.

This is because action(R)=print (or write) and type(R)=printer (or filestore) evaluate to false, and so ¬ evaluates to true.

The policy for the OR instance owned by Cambridge is refined to become

role(S)=student∧location(R)=UK∧
action(R)=access∧type(R)=or∧
Time≥9:00∧Time≤18:00

for the same reason as above except in this case the location is indeterminate.

The policy for the printer type is

role(S)=student∧location(R)=UK∧action(R)=print ∧
type(R)=printer∧¬(print{Copies}>2)))∧
Time≥9:00∧Time≤18:00.

This is because we simplify the expression action(R)=print∧type(R)=printer∧(¬(action(R)=print∧ type(R)=printer∧print{Copies}>2)) to the one above, and owner(R)=man is false so this sub-expression evaluates to true. The policy for the printer instance is FALSE because its location is not the UK.

The policies for type filestore can be produced from the high level policy

$role(S)=student \wedge location(R)=UK \wedge$
$action(R)=write \wedge type(R)=filestore \wedge$
$(\neg(owner(R)=man \wedge write\{Size\}>1MB)) \wedge$
$Time \geq 9{:}00 \wedge Time \leq 18{:}00.$

The policy for the filestore instance at Oxford is

$role(S)=student \wedge action(R)=write \wedge$
$type(R)=filestore \wedge Time \geq 9{:}00 \wedge Time \leq 18{:}00.$

Whilst the policy for the filestore instance at Manchester is

$role(S)=student \wedge action(R)=write \wedge$
$type(R)=filestore \wedge$
$(\neg(write\{Size\}>1MB)) \wedge Time \geq 9{:}00 \wedge Time \leq 18{:}00.$

We hope it is obvious to the reader what the policies for the remaining types and instances will be.

7. Conclusion and Future Work

By considering the multiple resource types of a distributed application as a single abstract resource type, it is possible to write a resource type hierarchy to describe how the abstract resource type is comprised of the multiple distributed resource types. A resource type hierarchy can then be used to describe a resource instance. Policy decomposition is achieved by refining an overall high level policy into a set of low level policies for the concrete resource types and then by refining the policy for a resource type into the policy that is specific to a resource instance.

The proposed policy refinement approach is appropriate because it ensures that the low level policies are in compliance with the high level policy and are simpler than the high level policy in terms of the number of policy components contained in them.

We suspect that the current policy simplification process is still incomplete because we can only identify syntactically identical expressions in order to remove redundancy from a refined policy.

At the moment, we use Protégé as the policy editor, which uses OWL as the policy language to define the high level policies and resource type hierarchies. A backend compiler is proposed, which will translate the generated low level policies in the ALET into current PDP policy languages such as XACML and PERMIS.

A distributed access control infrastructure needs to distribute the refined policies to specific PDPs, and to co-ordinate decision making when the concrete resources are alternatives for each other. The next steps are to define and implement the distribution and co-ordination protocols and integrate these into an existing policy based access control system such as PERMIS.

8. References

[1] J. Mendling, M. Strembeck, G. Stermsek, and G. Neumann, "An Approach to Extract RBAC Models from BPEL4WS Process", *Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2004),* Modena, Italy, 14–16 June 2004, pp. 1-6.

[2] D.W. Chadwick and A. Otenko, "The PERMIS X.509 Role Based Privilege Management Infrastructure", *Future Generation Computer Systems,* Elsevier Science Publishers B. V., Feb. 2003, pp. 277-289.

[3] W. Johnston, S. Mudumbai and M. Thompson, "Authorization and Attribute Certificates for Widely Distributed Access Control," IEEE 7th Int Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE), Stanford, CA. June, 1998, pp. 340 - 345 (see also http://www-itg.lbl.gov/security/Akenti/)

[4] A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo, "A Goal-based Approach to Policy Refinement", *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04),* Yorktown Heights, New York, 07 – 09 June 2004, pp. 229-239.

[5] N. F. Noy, M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, and M. A. Musen, "Creating Semantic Web Contents with Protege-2000", IEEE Intelligent Systems, 2001, 16(2), pp. 60-71.

[6] M. Dean, D. Connolly, F. Harmelen, J. Hendler, I. Horrocks, D.L.McGuinness, P.F. Patel-Schneider, L.A. Stein, "OWL Web Ontology Language 1.0 Reference", *W3C Working Draft,* 29 July 2002.

[7] S. Godik and T. Moses, "eXtensible Access Control Markip Language (XACML) Version 1.0", 18 Feburary 2003, http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf

[7] J. Park, X. Zhang, and R. Sandhu, "Attribute Mutability in Usage Control", *IFIP TC11/WG11.3 Eighteenth Annual Conference on* Data *and Applications Security,* Sitges, Catalonia, Spain, 25-28 July, 2004, pp.15-29.

[8] V. Welch, F. Siebenlist, D. Chadwick, S. Meder and L. Pearlman. "Use of SAML for OGSA Authorization", June 2004, Available from https://forge.gridforum.org/.