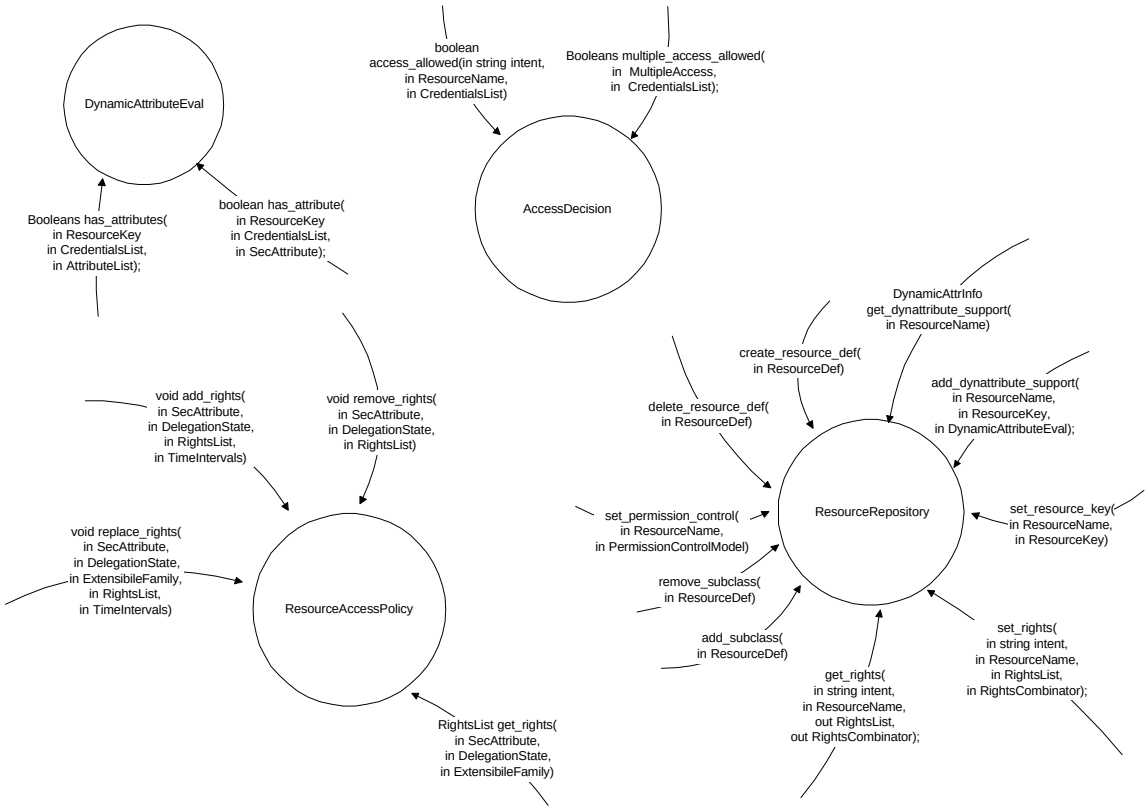
Carol Burt
2AB, Inc.
08/26/98

Hi,

Sorry we missed the call... this doc I think answers any homework we had... and a bit more. The following IDL is the basis of what 2AB would like to propose as a foundation for the HRAC RFP submission. I think it is complementary to Bob Blakley's model and the idl he submitted, although there are differences we need to resolve. I have added comments inline to the IDL. One noticeable difference is that I did administrative interfaces - it looks like from the minutes of the conference call that some submitters want to avoid this?

The complete unadorned IDL is available and I'll send it with this doc. As presented here, this IDL will compile with Orbix 2.3c compiler(which we know is zero guarantee that it is legal ;-)

To understand this proposal, Security Service 1.2 (ptc/98-01-02) probably needs to be nearby. The model I propose is [I think - I'm no expert] consistent with the access model in CORBA Security, but extends it (not directly, but such that the model is consistent) in some important ways for resource access control. To save you some time if you don't have 1.2 handy, I've cut the pdf file down to a few relevant pages and a few surrounding ones and will attach it to this proposal in pdf format. I also have the Security 1.2 idl I will forward as well if anyone needs it. The IDL below is in BLUE and the explanatory text in black - sorry if you don't have a color printer. I'm sure there are numerous logic flaws... I'm looking forward to all of you finding them. :-)
A [tiny] visual is inserted here for those who like pictures like me.



Carol Burt
2AB, Inc.
08/26/98

```
//============================================================================
//File: DfResourceAccessControl.idl

//

#ifndef _DF_RESOURCE_ACCESS_CONTROL_IDL_
#define _DF_RESOURCE_ACCESS_CONTROL_IDL_

#include <orb.idl>

#include "Security.idl"
#include "SecurityLevel2.idl"
```

This proposal uses Types defined in both Security.idl and SecurityLevel2.idl

```
// #include "NamingAuthority.idl"
```

This proposal does not use the NamingAuthority.idl file directly (this is used in all CORBAMed proposals, however, it does require that the ResourceName for resources that are BASECLASS resources - see below - must be a NamingAuthority::QualifiedNameStr for interoperability with other CORBAMed specifications that use this module. We can talk about strengthing this requirement to make the string a union type, but I thought that overkill.

```
#pragma prefix "omg.org"



module DfResourceAccessControl
{

interface DynamicAttributeEval;              // forward

//*********************************************************
//         Basic Types
//*********************************************************

typedef SecurityLevel2::CredentialsList CredentialsList;
typedef Security::AttributeList AttributeList;
typedef Security::SecAttribute SecAttribute;
typedef Security::DelegationState DelegationState ;
```

The types below are the Security types that this proposal is dependent on. I believe this spec could easily be used by applications where CORBASec is not available -- they simply would have to create these types. One question is whether to pass CredentialsList or to pass AttributeList. The question is whether an application that isn't CORBASec protected would want to create the Credential objects?

```
enum PermissionControlModel {GRANTED_RIGHTS, DENIED_RIGHTS};
```

Carol Burt
2AB, Inc.
08/26/98

This is an enhancement to the access control model in CORBASec. On a resource basis, the model can be set to a “granted_rights” model or a “denied_rights” model. This would mean that the enterprise user could configure based on explicitly granting rights, or explicitly denying rights for a SecAttribute -- in fact both types of rights could be assigned and depending on the resource the AccessDecision object would use the proper set of rights.

```
typedef sequence<boolean> Booleans;

typedef sequence<octet> ResourceKey;
```

This ResourceKey is defined by the application and is used when AccessDecision needs to call back to the application to determine if a dynamic attribute is true. It is opaque to the HRAC.

```
typedef string ResourceType;


const ResourceType BASECLASS = "BASECLASS";
const ResourceType SUBCLASSTYPE = "SUBCLASS";

const ResourceType  ATOMICTYPE = "ATOMIC";
```

Resources are of 3 types. A base type is basically the root of a tree, Subclass types may be linked to base types or other subclass types. This is your traditional file system approach with the exception that the subclass type can be a leaf. If such a tree is defined, the required rights associated with the subclasses down the tree must be subclasses of the required rights of the parents. An atomic resource stands alone and is not associated with any tree structure.

```
struct ResourceNameComponent {

      string                 id;
      ResourceType           kind;
};
typedef sequence<ResourceNameComponent> ResourceName;
```

ResourceName is a sequence of structures that includes a name (id) and a ResourceType as shown above. This is structured for ease of integration with a Naming Service if a vendor wished to do something like that.

If the ResourceType is “BASECLASS”, then the string must be a NamingAuthority::QualifiedNameStr

```
struct DynamicAttrInfo {
      DynamicAttributeEval          evaluator;
      ResourceKey                   key;
      AttributeList                 dyn_attrs;
};
```

This structure is used to provide information about the dynamic attributes and how to resolve them. (Note that DynamicAttributeEval is an interface that provides this service).

```
struct ResourceDef {

      ResourceNameComponent   local_name;      // Name & Kind

      PermissionControlModel  control_model;   // Denied/Granted rights

      boolean                 supports_dynattr;

};
```

The ResourceDef includes information necessary to do access control on a resource. See the ResourceRepository interface.

```
//typedef Security::ExtensibleFamily ExtensibleFamily;

//typedef Security::RightsCombinator RightsCombinator;



struct ExtensibleFamily {

      unsigned short    family_definer;

      unsigned short    family;
};

enum RightsCombinator {
      SecAllRights,
      SecAnyRight
};
```

They types above could be typedef'd... they are included here without typedef to keep the Idl easy to read without referring back to the Security spec excessively. They have the exact same meaning as in the security spec.

```
// extension of Security::Right...



struct Right {
```

```
      PermissionControlModel control_model;

      ExtensibleFamily  rights_family;

      string            right;

};

typedef sequence<Right> RightsList;
```

Note that this is NOT the identical structure as a right in the security specification although it basically is used the same. It has been extended to allow a right to be assigned as a GRANTED_RIGHT or a DENIED_RIGHT. This is to support both implicit and explicit models of access control.

```
typedef TimeBase::IntervalT IntervalT;

typedef sequence<IntervalT> TimeIntervals;




//*************************************************************

//     interface ResourceAccessPolicy

//*************************************************************
```

This interface is similar to the DomainAccessPolicy interface in CORBASec. The difference is that Domain Access is based on granting rights and revoking rights. Since “rights” are based on a control model in this spec (GRANTED_RIGHT, DENIED_RIGHT), these operations add rights and remove rights (vs. grant/revoke). The right itself is a granted right or a denied right.

There is also a sequence of TimeIntervals which are the times when the right is not valid.

If you keep this in mind, you can follow the intent by reading about the DomainAccessPolicy (starting on p. 15-130) operations in CORBASec to understand the model.

```
interface ResourceAccessPolicy{


      void add_rights(

            in    SecAttribute        priv_attr,

            in    DelegationState       del_state,
            in    RightsList            rights,
```

```
            in    TimeIntervals           deny_times
      );

      void remove_rights(
            in    SecAttribute          priv_attr,
            in    DelegationState       del_state,
            in    RightsList            rights
      );

      void replace_rights (
            in    SecAttribute        priv_attr,
            in    DelegationState       del_state,
            in    RightsList            rights,
            in    TimeIntervals           deny_times
      );

      RightsList get_rights (
            in    SecAttribute         priv_attr,
            in    DelegationState      del_state,
            in    ExtensibleFamily     rights_family
      );


};


//*****************************************************************
//     interface ResourceRepository
//*****************************************************************
```

This interface is a repository for the Resource information that is necessary for access decisions. The model is the same as the RequiredRights interface in CORBASec, but it has been extensively enhanced to include functionality necessary for resource control. In addition to setting the required rights, this allows you to define a resource of a certain type (BASECLASS, SUBCLASS, ATOMIC), define hierarchy's for the base/subclasses, provide the information necessary if access is control based on dynamic attributes. It also allows you to set the permission control used for the resource (GRANTED_RIGHTS, or DENIED_RIGHTS). Basically (and I'm not sure that there isn't a hole here quite frankly), the idea is that the rights assigned to the resource are compared with the rights assigned to the SecAttributes and the appropriate permission control (granted, denied) is applied. For granted rights this is identical to CORBASec use of rights... for denied rights I need to write up an explaination of how this works. See the DynamicAttributeEval interface of how this information is used.

```
interface ResourceRepository {


      void create_resource_def (

            in    ResourceDef resource);
```

```
      void delete_resource_def (

            in    ResourceDef resource);



      void add_subclass (in ResourceName parent,
                  in ResourceDef child);

      void remove_subclass (in ResourceName parent,
                  in ResourceName child);

      void add_dynattribute_support (
            in    ResourceName name,
            in    DynamicAttrInfo dynamic_info,
            in  AttributeList dyn_attrs);

      DynamicAttrInfo get_dynattribute_support(
            in ResourceName         name);

      void set_permission_control (
            in    ResourceName name,
            in    PermissionControlModel control_model);

      void set_resource_key (
            in    ResourceName name,
            in    ResourceKey key);

// The wrapper around RequiredRights functionality

      void set_rights(
            in  string                  intent,
            in    ResourceName          resource_name,
            in    RightsList            rights,
            in    RightsCombinator  rights_combinator
      );

      void get_rights(
            in  string                  intent,
            in    ResourceName          resource_name,
            out RightsList              rights,
            out RightsCombinator    rights_combinator
      );
};


//*****************************************************************
//     interface DynamicAttributeEval
//*****************************************************************
```

It is expected (but other options exist) that this interface will typically be implemented by the application itself or by the enterprise in which the application is installed. The idea is that the application (or some relationship service that is used by the application, or some local logic) is the only place that truly

understands the relationships that are necessary to determine if a Dynamic attribute is currently valid. For example, we may not want to or be able to “configure” the “ATTENDING_PHYSICIAN” SecAttribute. The application at runtime knows how to determine if this is true; the HRAC doesn’t. The application may use a relationship service, or it may “look” in the patient record, or it may know it is true some other way. So the idea is that for a Resource, a list of Dynamic Attributes that are part of the decision logic will be configured. These will include things like “ATTENDING_PHYSICIAN”. The application (or some other service -- internal or external to HRAC depending on implementation) will use the add_dynattribute_support () operation of the ResourceRepository to give the HRAC an object reference of an evaluator and a ResourceKey that can be used to ask if a [particular set of] attribute(s) is currently true. Note that this can be a relationship based attribute, but it isn’t restricted to that. It could be any SecAttribute that can be determined only by the application or by logic written for a particular installation.

The AccessDecision object would obtain an indication of whether dynamic attributes are supported and if it needed to determine if they are currently valid, it would obtain the DynamicAttributeEval and ResourceKey needed to call the DynamicAttributeEval and ask if the attribute is valid. Notice that rights are assigned to these attributes in the same way as any attribute... we just have to find out if they are currently part of the credentials.

```
interface DynamicAttributeEval {


      boolean has_attribute(

            in    ResourceKey     resource_key,
            in  CredentialsList existing_credentials,
            in    SecAttribute      dynamic_attr);


      Booleans has_attributes(

            in    ResourceKey     resource_key,


            in  CredentialsList existing_credentials,

            in    AttributeList     dynamic_attrs);

};
```

```
//*************************************************************
//     interface AccessDecision
```

Carol Burt
2AB, Inc.
08/26/98

```
//*****************************************************************
```

This is the AccessDecision interface. I'm requesting that we not use the term "operation" as an identifier for the first parameter. This is because it can be confused with an operation on a CORBA interface. This is clearly one way that the parameter might be used, but this type of operation might also be considered a subclass resource and the operation would be "use". And it has little meaning when the resource is actually a parameter of a corba operation... anyway, I'm suggesting that this word be changed to "intent"... to clearly indicate that we expect it to be (but obviously don't require) things like "read, write, update, use, ... I think I could actually make an argument that it isn't needed at all (or should be a "requested_right")... but I haven't thought about that enough.

```
interface AccessDecision {


      struct AccessDef {
            string                intent;
            ResourceName      name;
      };
      typedef sequence<AccessDef> MultipleAccessDef;

      boolean access_allowed(
            in  string              intent,
            in    ResourceName    name,
            in  CredentialsList credentials);

      Booleans multiple_access_allowed(
            in  MultipleAccessDef   requested_access,
            in  CredentialsList credentials);


};
};
#endif  // DfResourceAccessControl
```