

---

# DAF Extensions for Updates

---

*An Informal Proposal*  
*Arnold deVos*  
*adv@langdale.com.au*

---

## 1. Introduction

---

This short paper shows how the Utility Management System Date Access Facility (DAF) can be extended to meet the update requirements of Common Data Access Phase II (CDA II).

The DAF is at the revised submission stage in the Object Management Group and will be considered for adoption in November. It is also the basis of a draft IEC proposal.

Discussion on requirements for CDA II has been in progress for some time. It is not the intention here to add to the requirements discussion.

The aim is merely to propose an implementation of those requirements that preserves the design integrity of the DAF. Most of the considerations are technical or concern style. Some only apply to the CORBA interpretation of CDA.

---

---

## 2. Summary of Requirements

---

---

The following list of update requirements has been gleaned from the CDA discussion and the relevant CCAPI and IEC documents. Any additions or changes to those requirements are unintentional and should be corrected in future drafts. The list is strictly a summary and the original documents should be read for complete information.

1. **Update of objects.** In DAF terms, this is the ability to update the property values of extant resources.
2. **Control of object lifecycles.** Provide the ability to create and destroy objects accessible through CDA. In DAF terms, this is the ability to create and delete resources. This also implies associating and disassociating URI's with resources.
3. **Update of multiple objects.** Provide the ability to update a collection of objects in one operation.
4. **Update of metadata.** Create, delete and update data definitions. In DAF terms, this means creating, deleting and updating resources of type `rdf:Class` and `rdf:Property`.

In addition to the forgoing CDA II requirements there are some requirements from the DAF RFP (and CDA I) which must be observed.

5. **Wrap extant systems.** It must be possible to implement the interfaces for a variety of systems including proprietary databases and existing EMS systems. In
6. **Multiple clients, providers and concurrency.** Provision must be made for multiple clients and multiple data providers. When updates are concerned, this is requirement is interpreted to imply a form of concurrency control.

### 2.1 Requirements Not Covered

Not all requirements proposed for CDA II are in the update area. Some of the others can be described as:

1. Expanded query facilities to deal with ad-hoc collections of resources.
2. Expanded event facilities to transmit the changed data in the notification.

Solutions are not proposed for these requirements in the current paper. It is expected that a proposal will be developed paralleling the OLE for Process Control (OPC) capabilities in this area. Such a proposal might include the ability to define arbitrary sets of resources that are then used to subscribe for update notifications.

The present proposal interacts with that type of facility only indirectly (via the proposed concurrency control mechanism).

---

---

## 3. Design Rationale

---

---

A number of design considerations fall outside the functional requirements. These include:

- The module structure of the DAF.
- The use of Uniform Resource Identifiers (URI's).
- The use of common interfaces for data and metadata.
- The style of interface design and the idioms employed.
- Solutions for concurrency control

The last of these points requires special mention.

### 3.1 Concurrency Control

An essential contribution of this proposal is a concurrency control mechanism that is suitable for a wide range of data providers, including those that do not support transactions.

#### 3.1.1 Design Forces

The real time databases underlying the majority of EMS systems in service today do not support transactions. Interface designs that rely on "ACID" transactions cannot be implemented correctly over these systems.

Furthermore, many of these systems provide only coarse-grained locking facilities, which work at the whole database, table, or physical partition level. Implementations that hold locks of this type on behalf of clients incur severe performance penalties. Deadlocks are likely in an environment with more than a few clients.

The interface design should enable simple implementations that are not prone to the foregoing problems.

#### 3.1.2 Solution Approach

The present proposal relies on a system of preconditions to check concurrency conflicts. This solution is similar to some optimistic locking strategies, is scaleable and is simple to implement.

Each update operation is accompanied by precondition information that the server uses to detect conflicting concurrent updates. The precondition information represents the client's assumptions about the data, at the time the update was formulated.

If these assumptions hold true later, at the time the server applies the update, then no conflicting updates have intervened. Conversely, if the assumptions no longer hold the updates must be rejected. This is equivalent to a lock conflict.

The scope of the preconditions accompanying an update is controlled by the client. This is equivalent to the scope of the read-lock in a locking scheme.

The granularity of the precondition data is at the property level. This is equivalent to the lock granularity.

The data provider is required to lock out other sources of updates in the short period while an update operation is executed. However, this period is far shorter than the time taken by the client to formulate the update and does not include any network latency.

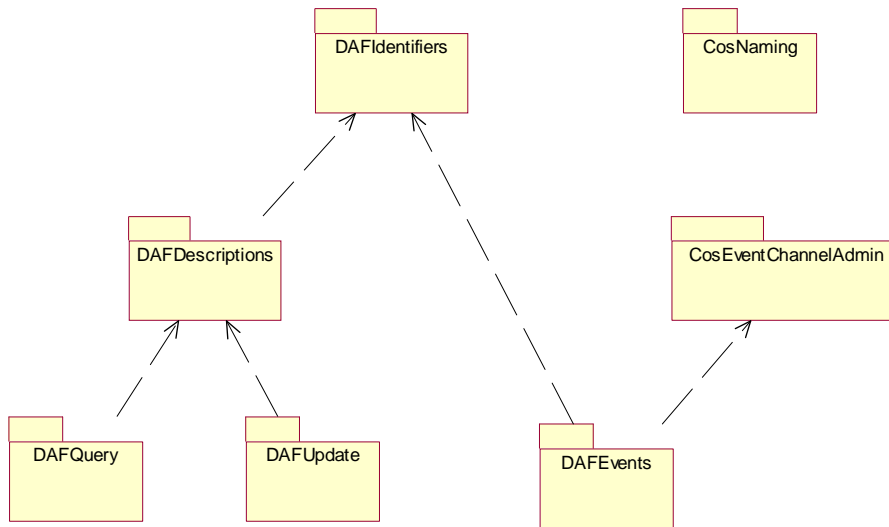
The data provider is not required to maintain lock state information on behalf of the clients between update operations. There are no abandoned locks to be timed out, nor deadlocks to be detected and cleared.

---

## 4. DAFUpdate Module

---

The additional definitions for updates are placed in a new module, DAFUpdate. This simplifies the specification and aids compatibility between CDA and CDA II implementations. The following module diagram shows how DAFUpdate fits with the existing DAF modules.



---

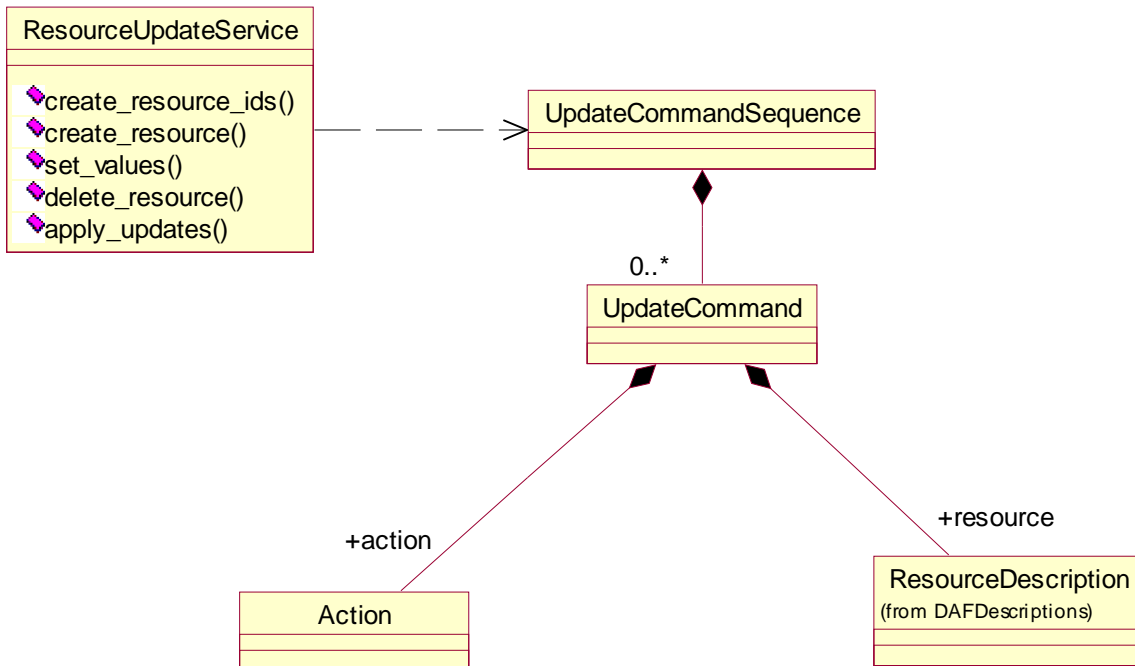
---

## 5. ResourceUpdateService

---

---

The update functionality is provided by a new service, ResourceUpdateService. This is defined as an interface within the DAFUpdate module. The following class diagram shows the new service and its associated structures.



### 5.1 ResourceUpdateService Interface

The ResourceUpdateService interface provides operations to update data that is accessible via the DAF.

- create\_resource(), set\_values() and delete\_resource() operations create, update and delete individual resources. This includes CIM-defined resources, other application-specific resources, and the metadata resources corresponding to classes and properties.
- The create\_resource() and set\_values() operations also include the ability to associate a URI with a resource or change the URI for a resource. The property <http://omg.org/schema/DAF#URI> is provided for this.
- The create\_resource\_ids() and apply\_updates() operations enable a sequence of updates to be performed in a single operation. The sequence may involve an arbitrary group of resources. If new resources are created, they can be assigned ResourceID's allocated in advance by create\_resource\_ids().

Note that the interface provides two ways to effect any given update. The create\_resource(), set\_values() and delete\_resource() operations are provided for simple, single-shot updates. The more general apply\_updates() operation can perform the same functions but can also carry out multiple updates.

## 5.2 create\_resource()

This operation creates a single resource and optionally allocates a ResourceID for it.

```
ResourceID create_resource( in ResourceDescription new_resource )
    raises( PreconditionFailed, UpdateError );
```

The `new_resource` argument is a description of the resource to be created. The operation returns the resource identifier of the new resource.

If the `id` member of `new_resource` is null, a ResourceID is allocated for the new resource. Otherwise, if `id` is not null, it must be a ResourceID returned from `create_resource_ids()` and must not designate an extant resource. If the designated resource exists, a `PreconditionFailed` exception is raised and no resource is created.

### 5.2.1 Initial Property Values

The `new_resource` argument also provides initial property values for the resource. The `values` member of `new_resource` contains this information. Any properties not represented in the `values` member are given implementation-defined initial values.

### 5.2.2 Type of Resource Created

A value for the `type` property (<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222#type>) must be supplied in the `values` member. It specifies the class to which the new resource will belong. (This determines the table to store the resource information in the underlying database in many implementations).

This value is a ResourceID and is typically obtained from the class name (URI) via the `ResourceIDService`.

### 5.2.3 Assigning a URI

If the `URI` property (<http://omg.org/schema/DAF#URI>) is supplied, then the given URI is allocated and associated with the new resource. This URI will be returned by the `get_uris()` operation of the `ResourceIDService`. If the `URI` property is not supplied, the data provider will allocate a URI for the resource (possibly based on its ResourceID).

### 5.2.4 Creating Metadata

The `type` property indicates whether a metadata item is being created. The new resource will be a class if the `type` is “RDF class” (<http://www.w3.org/TR/1999/PR-rdf-schema-19990303#Class>).

A new `Property` is created when the `type` is “RDF property” (<http://www.w3.org/TR/1999/REC-rdf-syntax-19990222#Property>).

## 5.3 set\_values()

This operation assigns values to the properties of a single resource.

```
void set_values(
    in ResourceDescription precondition,
    in ResourceDescription resource )
    raises( PreconditionFailed, UpdateError );
```

The resource argument specifies the update. Its id member designates the resource to be updated. Its values member lists the properties to be assigned and their new values.

### 5.3.1 Preconditions

The precondition argument specifies the initial state of the resource, prior to update. If the resource is not in this state, a `PreconditionFailed` exception is raised and no property values are modified. This guards against a conflicting, concurrent update of the same resource.

The id member of the precondition argument must agree with the id of the resource argument. If the designated resource does not exist, `PreconditionFailed` is raised.

The values member of the precondition may be empty, or contain any combination of properties defined for this resource. If it is not empty, the given property values are compared to the corresponding properties of the resource. If any disagree then `PreconditionFailed` is raised.

## 5.4 delete\_resource()

This operation destroys a single resource.

```
void delete_resource( in ResourceDescription resource )
    raises( Preconditionfailed, UpdateError );
```

The resource argument designates the resource to be deleted via its id member.

After the operation returns, the resource will not be accessible via the DAF interfaces. That is, it will not appear in query results and neither its URI nor its ResourceID will be translated by the ResourceIDService.

However, its ResourceID will not be reused for any other resource for some time T, which is implementation dependent but must be at least as long as the client maintains a valid reference to the DAF service.

### 5.4.1 Preconditions

The resource argument specifies the initial state of the resource, prior to deletion. If the resource is not in this state, a `PreconditionFailed` exception is raised and it is not deleted. This guards against a conflicting, concurrent update of the same resource. The check is performed in the same way as the precondition argument of `set_values()` (above).

## 5.5 create\_resource\_ids()

This operation creates one or more ResourceID values which can be used in subsequent `create_resource()` operations or `UpdateCommand CREATE_RESOURCE_ACTION`'s.

```
ResourceIDSequence create_resource_ids(
    in ResourceID prototype,
    in int how_many )
    raises ( UnknownResource );
```

The return value is a sequence of valid ResourceID values. None of the resource identifiers designates an extant resource at the time it is returned. None of identifiers returned is equal to any previously returned identifier.

The prototype is the resource identifier of an existing resource. The new resource identifiers will be managed by the same data provider as the prototype. A null resource identifier may be provided in which case the implementation will select a data provider.

The integer, `how_many` indicates the number of resource identifier values to be produced. The operation will return a sequence of at most this many resource identifier values. (If less than `how_many` are returned then the client may make another request.)

## 5.6 `apply_updates()`

This operation performs a series of updates as a single operation.

```
void apply_updates( UpdateCommandSequence updates )
    raises( PreconditionFailed, UpdateError );
```

The `updates` argument is a `UpdateCommandSequence`, which is defined in the next section. The operation interprets this sequence, enforces the implicit or explicit preconditions and performs the specified actions.

If any precondition is found not to hold at the point the operation is invoked, the `PreconditionFailed` exception is raised and no updates are performed. This guards against conflicting concurrent updates.

## 5.7 `UpdateCommandSequence`

The `UpdateCommandSequence` represents a series of updates to be performed in order, as a batch, by a single `apply_updates()` operation. Each `UpdateCommand` in the sequence contains an action and a resource description. The action member specifies what is to be done and how to interpret the corresponding resource description. The actions are:

**CREATE\_RESOURCE\_ACTION** - creates a resource from the resource description. The behavior is identical to the `create_resource()` operation, except no `ResourceID` is returned. (See the `create_resource()` operation, above.) If the client needs to refer to this resource explicitly, either later in the `UpdateCommandSequence` or in a subsequent operation, it must provide a valid `ResourceID` in the resource description. (New `ResourceID`'s are obtained from a `create_resource_ids()` operation, see above). If a non-null `ResourceID` is given but it designated an extant resource, a `PreconditionFailed` exception is raised.

**SET\_VALUES\_ACTION** - updates the property values of a resource. The resource, the properties and their new values are given in the resource description. The effect of a combination of a `SET_VALUES_ACTION` and a `ENFORCE_PRECONDITION_ACTION` is identical to the `set_values()` operation.

**DELETE\_RESOURCE\_ACTION** - deletes the resource specified in the resource description. The behavior is identical to the `delete_resource()` operation. If the specified resource does not exist or its properties do not agree with the resource description, a `PreconditionFailed` exception is raised.

**ENFORCE\_PRECONDITION\_ACTION** - checks that the given resource description is accurate. If the specified resource does not exist or its properties do not agree with the resource description, a `PreconditionFailed` exception is raised.

## 5.7.1 Preconditions

Preconditions are the basis of concurrency control when more than one client is performing updates. Each precondition is an assumption on the part of the client about the state of resources prior to update. A precondition failure indicates that another client has completed a conflicting update.

A precondition can be explicitly specified by the `ENFORCE_PRECONDITION_ACTION`. As described above, the resource is checked to ensure it exists and matches the given resource description. Explicit preconditions can be enforced for any resources on which the present update sequence depends. This includes but is not limited to the resources being modified.

The resource description in a `DELETE_RESOURCE_ACTION` is similarly treated as an implicit precondition.

The `CREATE_RESOURCE_ACTION` carries an implicit precondition that the designated resource does not already exist.

All preconditions are enforced before any updates are performed. In other words, the values in a `ENFORCE_PRECONDITION_ACTION` or `DELETE_RESOURCE_ACTION` are compared to the initial state of the designated resources no matter where they appear in the `UpdateCommandSequence`. (Incidentally, this implies that a resource cannot be created and deleted in the same `UpdateCommandSequence`.)

Furthermore, if a `PreconditionFailed` exception is raised, then none of the updates is performed.

---

## 6. Complete IDL

---

```
//File: DAFUpdate.idl
#ifndef _DAF_UPDATE_IDL_
#define _DAF_UPDATE_IDL_

#include <DAFDescriptions.idl>

#pragma prefix "omg.org"
module DAFUpdate
{
    // types imported from the descriptions and identifiers modules
    typedef DAFDescriptions::ResourceDescription ResourceDescription;
    typedef DAFIdentifiers::ResourceIDSequence ResourceIDSequence;

    // An Action is is the "verb" for an UpdateCommand structure.
    typedef short Action;
    const Action CREATE_RESOURCE_ACTION = 1;
    const Action SET_VALUES_ACTION = 2;
    const Action DELETE_RESOURCE_ACTION = 3;
    const Action ENFORCE_PRECONDITION_ACTION = 4;

    // an Update consists of a command and its associated data.
    struct UpdateCommand
    {
        Action action;
        ResourceDescription resource;
    };
    typedef sequence<Update> UpdateCommandSequence;

    // updates can raise these exeptions
    exception PreconditionFailed { string reason };
    exception UpdateError { string reason; };
    exception UnknownResource{ string reason; };

    // the update service
    interface ResourceUpdateService
    {
        ResourceIDSequence create_resource_ids(
            in ResourceID prototype,
            in int how_many )
            raises ( UnknownResource );

        ResourceID create_resource( in ResourceDescription new_resource )
            raises( PreconditionFailed, UpdateError );

        void set_values(
            in ResourceDescription precondition,
            in ResourceDescription resource )
            raises( Preconditionfailed, UpdateError );

        void delete_resource( in ResourceDescription resource )
            raises( Preconditionfailed, UpdateError );

        void apply_updates( UpdateCommandSequence updates )
            raises( Preconditionfailed, UpdateError );
    };
};
#endif // _DAF_UPDATE_IDL_
```