

Atacama Large Millimeter

Revision: 1.0

2005-105-17

Documentation

Alessandro Caproni

Configuration DataBase CDB

Software Specification

Alessandro Caproni (acaproni@eso.org)

European Southern Observatory

Steve Harrington (sharring@nrao.edu)

National Radio Astronomy Observatory

Keywords: CDB DOC

Author Signature:

Date:

Approved by:

Signature:

Institute:

Date:

Released by:

Signature:

Institute:

Date:

Table of Contents

1 Introduction.....	6
1.1 Reference Documents.....	7
1.2 Acronyms.....	7
2 Basics.....	8
3 Structure of the CDB.....	9
4 DAL.....	11
4.1 DAO.....	14
4.2 Adding user defined data to a component.....	15
4.2.1 Mapping to XSD.....	16
5 Modifying the data: WDAL and WDAO.....	21
6 cdbjDAL	24
7 Command line utilities for dealing with the CDB.....	26
7.1 cdbjDALClearCache command	26
7.2 cdbjDALShutdown command	27
7.3 cdbRead command	27
7.4 CDB Browser	28
7.5 CDB Checker.....	28
8 Resolving the Configuration Database Reference.....	29
9 Manager Configuration Database.....	29
10 Container Configuration Database	30
11 Component Configuration Database.....	31
12 Characteristic Component Configuration Database.....	31
13 Alternative CDB structure for component's deployment.....	32
14 Hierarchical components and CDB structure.....	33
14.1 Components.xml	35
14.2 Hierarchical components as directory tree	35
14.3 Hierarchical components as single file	36
14.4 Deep hierarchy with pure-logical nodes	36
14.5 Hierarchy with pure-logical nodes and sub-nodes in one file	37
14.6 Putting multiple XML files in the same directory using XInclude	37
14.7 Deployment of Dynamic Components in multiple files.....	39
Appendix A. Obsolete CDB Table interfaces.....	40

1 Introduction

The Configuration Database (CDB) in ACS serves as a repository for a variety of configuration data that is needed by the ALMA system. Information contained in the CDB includes configuration and deployment information for Components, Containers, and the ACS Manager. This configuration and deployment information consists of data needed during startup or at runtime for the ALMA system.

In its basic form, the CDB is a set of simple text files (e.g. XML instance documents and XSD schema documents) that are used to configure the system. Because ALMA is a distributed system, with various pieces running on different computers simultaneously, it would be costly, confusing, and a maintenance nightmare to require such configuration files on every machine in the system. By using a single CDB, maintenance of configuration data is much simpler. At the same time, the CDB supports development, testing, and configuration of components in the system, along with the ability to easily and transparently switch between development and production configurations.

As noted, the CDB can be used to store configuration settings for Components, Containers, and the Manager. There can be many types of Components in the system and for each type of Component there can be potentially many instances. Typically¹, the number of Component instances which will exist in the system and how those instances will be configured is stored in the CDB. Similarly, there can be many Containers in the system, and these can be configured in the CDB. While there will often be only a single Manager, it is possible to have multiple Managers in certain contexts; in either case, various settings for the Manager(s) can be configured in the CDB.

For programmers, the CDB API allows both remote and local C++, Java, and Python processes to read and write the configuration database. In addition, a set of utilities allows the user to manipulate, browse, and validate the configuration database:

- CDB Browser is a java process to read and write the database through a Graphical User Interface [RD07].
- CDB Checker: is a tool to check consistency and syntactical correctness of the CDB
- cdbRead: reads a record writing the output to stdout
- cdbjDALClearCache: clears the cache of the cdb
- cdbjDALShutdown: shuts down the CDB

¹In some cases, e.g. dynamic components, it may not be the case that each instance is individually configured in the CDB; rather, a single CDB configuration entry may suffice for one to many instances of such components.

The directory \$ACSDATA/config/defaultCDB contains a sample configuration database. This is the configuration database that is used by default when ACS is installed. This database declares a number of Component instances that are defined in the ACS example module, acsexmpl, as well as the Java and Python example modules (jcontexmpl and acspyexmpl).

These Components are provided as examples for the users of ACS and for testing ACS. It is suggested to make a backup copy of defaultCDB and to modify your own copy. It is also a good practice to create new Configuration Databases for specific applications, assigning them explicit names, like corrCDB, instead of using defaultCDB all the time. Each of the items mentioned above will be described in more detail in the following sections.

1.1 Reference Documents

[RD01] Basic Control Interface Specification, M.Plesko, G.Tkacik, G.Chiozzi -

(http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACS_Basic_Control_Interface_Specification.pdf)

[RD02] Adaptive Communications Environment (ACE) -

(<http://www.cs.wustl.edu/~schmidt>)

[RD03] ALMA Common Software Technical Requirements - COMP-70.25.00.00-003-A-SPE, G.Raffi, B.Glendenning, J.Schwarz

(<http://www.eso.org/~almamgr/AlmaAcs/MilestoneReleases/Phase1/ACSTechReqs/Issue1.0/2000-06-05.pdf>)

[RD04] ALMA Common Software Architecture, G.Chiozzi, H. Sommer -

(<http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACSArchitectureNL.pdf>)

[RD05] XML Schema files -

(http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACS_docs/schemas/index.html)

[RD06] C++ Component/Container Framework Tutorial

(http://www.eso.org/projects/alma/develop/acs/OnlineDocs/BACI_Device_Server_Programming_Tutorial.pdf)

[RD07] Configuration Database Browser User's Manual

(http://www.eso.org/projects/alma/develop/acs/OnlineDocs/CDB_Browser_Users_Manual.pdf)

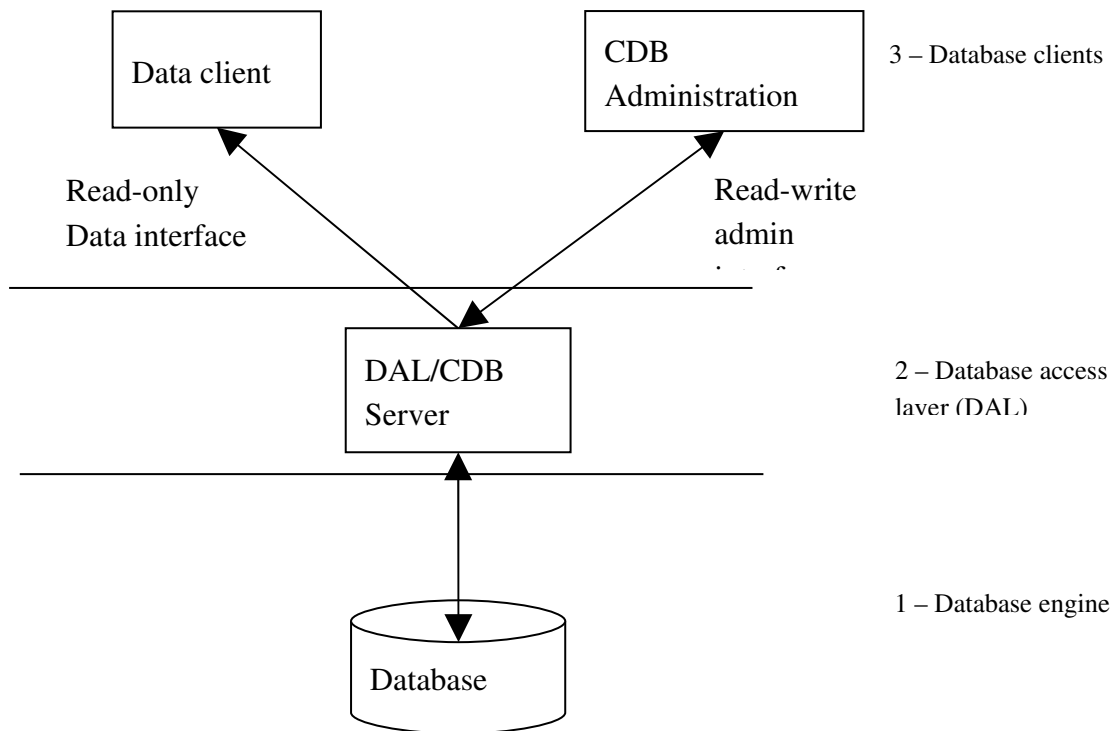
1.2 Acronyms

ACS	ALMA Common Software
CDB	Configuration Database
DAL	Data Access Layer
DAO	Data Access Object
IOR	Interoperable Object Reference
POA	Portable Object Adapter
XML	Extensible Markup Language
XSD	XML Schema Definition

2 Basics

The actual implementation of the database consists of a set of XML files parsed against a set of XSD files. The user edits these files entering the configuration data; a text editor, an XML editor, or the cdbBrowser can be used for this purpose. The different configurations can then be stored using version control software, such as CVS, if desired.

The CDB is a three-tiered architecture including: database clients, a data access layer, and a database engine.



Data clients can be Components (that read the CDB) as well as tools like the `cdbBrowser`. Data clients can also write the CDB, changing the actual content of the database, through an administrative interface. The `cdbBrowser` utility can also be used to edit the database.

The Data Access Layer (DAL) is the server (middle level) that replies to requests from clients (upper level), by querying the database (lower level). The interface to the database is the Data Access Object (DAO), which will be discussed in Section 4.1. The DAL server is started automatically as part of ACS (by the `acsStart` script or by the `acscommandcenter` graphical tool) and does not normally need to be started explicitly. The command to start the server is `cdjDAL` and it is described in section 6.

The database engine is independent of the upper and middle levels and can therefore be changed transparently. For example, the current implementation of the CDB consists of a set of XML and XSD files. However, the three-tiered architecture makes it possible to change the CDB engine in the future, if desired, in a transparent manner. For example, we could change the implementation of the CDB to a set of plain text files, a relational database, or some other mechanism, without requiring any changes to the upper layers.

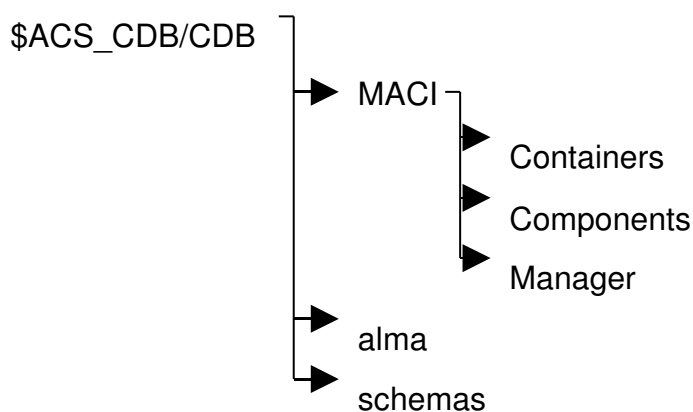
3 Structure of the CDB

The ACS Configuration Database (CDB) addresses ALMA's needs for defining, accessing, and maintaining the configuration of the system. This configuration consists of a set of parameters that have to be configured in a (quasi) persistent store and read at run time. Information

contained in the CDB includes configuration and deployment information for Components, Containers, and the Manager. This configuration and deployment information consists of the data needed during startup or at runtime.

There can be many types of Components in the system and for each type of Component there can be potentially many instances. Typically², the number of Component instances which will exist in the system and how those instances will be configured is stored in the CDB. Similarly, there can be many Containers in the system, and these can be configured in the CDB. While there will often be only a single Manager, it is possible to have multiple Managers in certain contexts; in either case, various settings for the Manager(s) can be configured in the CDB.

The basic structure of the CDB is a directory tree containing various text files (XML, XSD) describing the system. The directory structure is:



As seen above, a standard configuration database contains three primary sub-directories:

- **MACI** – This directory contains configuration data for the Manager(s) and Containers, as well as the main deployment configuration file for Components. The Components' deployment configuration file contains information needed by the ACS Manager to determine which Container is responsible for hosting each Component, as well as where the actual implementation (logic) resides for each Component. Future examples will elaborate on these points.
- **alma** – This directory contains *configuration* data, in the form of XML files, that is required for certain types of Components in the ALMA system; some components require configuration information in the CDB, while others do not. For example, CharacteristicComponents (i.e. components with properties) always have configuration data in the CDB, but simple Components do not require such data. However, both simple

²In some cases, e.g. dynamic components, it may not be the case that each instance is individually configured in the CDB; rather, a single CDB configuration entry may suffice for one to many instances of such components.

Components and CharacteristicComponents require *deployment* information in the MACI directory (more specifically, in the file: MACI/Components/Components.xml – this will be explained in more detail later in this document, as well as alternate ways to organize this data). The XML files here are instance documents of the corresponding schemas contained in the schemas directory, described below.

- **schemas** – This directory contains schema (xsd) files which define a type of Component; each schema defines what individual instances of the Component type are required - or able in the case of optional information - to define in their XML instance documents (located under the alma directory tree, described above). The schemas can be used to define default values, max and min values, units, descriptions, etc. for the properties of CharacteristicComponents. They can also be used to add custom user-defined information to the CDB, as described in Section 4.2.

Each CharacteristicComponent instance in the Configuration Database is represented by an XML file in the alma folder. Each of those XML files is an instance of an XML schema (xsd) file located in the \$ACS_CDB/CDB/schemas directory³. When the XML instance document is parsed, the XML parser uses the corresponding XML schema to expand things like default values, etc.

As can be seen from the above discussion, each of these directories can contain various text files (XML, XSD) to describe the ALMA system configuration. Depending on what kind of Components and Containers exist in the system, different files may or may not be necessary. These ideas will be explored in more detail in Sections 12 and 13.

4 DAL

The Data Access Layer (DAL) is the definition of an API (application programmer's interface) for accessing CDB data programmatically; if you are only interested in how to configure the CDB by hand and do not need to programmatically query or manipulate data in the CDB, you may skip this section. The DAL is a three-tiered database access architecture. The first tier is the implementation of the CDB Data Access Object (DAO) interface, the second tier is the DAL server, and the third tier is data storage (i.e. the database). This three-tiered design maps directly to the three tiers discussed previously in Section 2, e.g. the DAO interface maps to (i.e. is used by) data clients, while the DAL server and database are as described in Section 2.

In the DAL architecture, data is accessed through DAOs. The DAO is a representation of one set of data (i.e. one record in DAL data storage). The DAO has a predefined interface, which is strongly typed. This means that data is fetched according to its type (long, double, string, etc.).

³ There are multiple possible locations for the schemas, see the discussion of the cdjDAL server in Section 6 for details.

Each DAO object can be local or remote; you can request the DAL server to create a DAO as a remote object or you can request data from the DAL server and create a DAO locally from that data. As a general rule, when you need small amounts of data it is much easier to create a DAO on the DAL server and get the required information from it. But if you need to make many requests on such a DAO, the network overhead will be significant. In such a situation, to save bandwidth and increase the performance, it is better to get the data from the server and build the object locally. Conceptually, you can imagine that the DAO object migrates from the remote machine to the local machine so that its functions are executed locally without network overhead.

The IDL interface for the DAL is as follows:

```
interface DAL {
    string get_DAO( in string curl )          raises (RecordDoesNotExist,XMLerror);
    DAO    get_DAO_Servant( in string curl ) raises (RecordDoesNotExist,XMLerror);
    oneway void shutdown();

    //data change handling
    long   add_change_listener( in DALChangeListener listener );
    void   listen_for_changes( in string curl, in long listenerID );
    void   remove_change_listener( in long listenerID );

    // listing
    string list_nodes( in string name );
};
```

As mentioned, to access the data in the configuration database, you have to get either a local or a remote DAO. To get a remote DAO you should call the `get_DAO_Servant` method passing the curl of the record you want to access. If you prefer to create the DAO locally, then you should call the `get_DAO` method, which returns an XML string representing the record that must be used to instantiate the local DAO.

The following C++ code is part of `acsexmplFilterWheel.cpp`⁴; it shows how to access a DAO (Java and Python snippets are also shown):

```
C++:

1 CDB::DAL_ptr dal_p = getContainerServices()->getCDB();
2 CDB::DAO_ptr dao_p = dal_p->get_DAO_Servant(m_fullName.c_str());

Java:

// within context of a try/catch block (see subsequent examples)
1 com.cosylab.CDB.DAL dal = getContainerServices().getCDB();
2 com.cosylab.CDB.DAO dao = dal.get_DAO_Servant(m_fullName);

Python:

0 import Acspy.Util.ACSCorba
```

⁴ This example is part of the `acsexmpl` module.

```
1 dal = Acspy.Util.ACSCorba.cdb()
2 dao = dal.get_DAO_Servant(m_fullName)
```

The first step is to get a pointer to the DAL (Java and Python do not use pointers, of course) from the ContainerServices as shown in line 1. Line 2 shows how to get a remote DAO from the DAL. The parameter `m_fullName` is the curl of the record being accessed. In this example we are accessing the XML describing the Component itself, so the `m_fullName` is “alma/FILTERWHEEL” (we will see later the structure of the CDB).

The following examples show how to create a DAO locally:

C++:

```
1 CDB::DAL_ptr dal_p = getContainerServices()->getCDB();
2 CDB::DAO_ptr dao_p = dal_p->get_DAO_Servant(m_fullName.c_str()); //Old
3 ACE_CString xml(dal_p->get_DAO(m_fullName.c_str()));
4 DAOImpl* dao_local = new DAOImpl(xml.c_str());
```

Java:

```
// within the context of a try/catch block; see other examples
1 DAL dal = getContainerServices()->getCDB();
2 DAO dao = dal.get_DAO_Servant(m_fullName); //Old
3 String xml(dal.get_DAO(m_fullName));
```

TODO - find out how to do this from Heiko

```
4 DAOImpl dao_local = new DAOImpl(xml);
```

Python:

```
1 import Acspy.Util.ACSCorba
2 dal = Acspy.Util.ACSCorba.cdb()
3 dao = dal.get_DAO_Servant(m_fullName) //Old
```

TODO - find out how to do this from David

```
4 String xml(dal.get_DAO(m_fullName));
5 DAOImpl dao_local = new DAOImpl(xml);
```

Line 1 and 2 are the same as before. Line three calls the `get_DAO` that returns the xml string representing the record of our component. Eventually, line 4 instantiates a `DAOImpl` object by passing the xml string from line 3 as parameter. The local `DAOImpl` object can be used in the same way as the remote object, but all the calls will be executed locally thereby reducing the network overhead. At the present the local DAO in C++ is not fully implemented.

4.1 DAO

The Data Access Object, as mentioned in the previous section, is an interface used to programmatically access the data of an individual record in the configuration database. The IDL interface of the DAO is shown below:

```
interface DAO : ACS::OffShoot {
    long      get_long( in string propertyName );
    double    get_double( in string propertyName );
    string    get_string( in string propertyName );
    string    get_field_data( in string propertyName );

    stringSeq get_string_seq( in string propertyName );
    longSeq   get_long_seq( in string propertyName );
    doubleSeq get_double_seq( in string propertyName );

    void      destroy();
};
```

The DAO interface contains typed methods which are used to access the data in the record; the user must be aware of the type of the data being read in order to invoke the appropriately typed method.

For example, the following lines show how to read the value of the `alarm_timer_trigger` attribute of the `cmdAz` property⁵ of the `mount` component (the full C++ example can be found in the `acsexmpl` module):

```
C++:
double att = dao_p->get_double("cmdAz/alarm_timer_trig");

Java:
// within context of a try/catch block (see subsequent examples)
double att = dao.get_double("cmdAz/alarm_timer_trig");

Python:
att = dao.get_double("cmdAz/alarm_timer_trig")
```

The DAO insulates user code from changes in the underlying database and DAL layers. It would be possible for ACS to change the implementation of the DAO (while leaving the DAO IDL interface unchanged), such that the implementation used, for example, a relational database; by leaving the IDL interface unchanged, user code using the DAO would not require any changes.

⁵ The value of that attribute is read from the XML (describing the component specified) the DAO obtained from the DAL. If this attribute is not explicitly defined in the XML, then it is read from the schema to the component (if there is a default value specified in the schema file).

4.2 Adding user defined data to a component

In the current implementation of the CDB, each XML file is parsed against a schema (i.e. XSD file). When creating a new `CharacteristicComponent` (i.e. a `Component` with properties), you have to define its xml description in the `alma` directory of the CDB. You must also write a schema file and place it in the `config/CDB/schemas` directory of the module for the `CharacteristicComponent`⁶. These steps are all described in the BACI programming tutorial [RD06].

There are situations when you need to add custom information to the XML description for a `Component` in the CDB. In this case you must define the schema accordingly. The `FilterWheel` and the `LampWheel` C++ examples, in the `acsexmpl` module in the ACS distribution, give two examples of such a situation.

The simplest situation shows the changes needed to add a simple attribute to the component's XML tag. This is shown in the `LampWheel` example. Suppose we have more than one lamp wheel in our system and we want to add a string describing the position of each wheel. We can add this description as an attribute of the tag for the component, e.g. `<LAMPWHEEL...>`. For this purpose we have to change both its `xsd` and `xml` files⁷.

In the schema (`xsd`) file, we have to add the description of the new attribute for the `LAMPWHEEL` tag specifying its name, `LampWheelDescription`, and its type, (`xs:string`). This is shown, below:

```
<xs:complexType name="LAMPWHEEL">
  ...
  <xs:attribute name="LampWheelDescription" type="xs:string" use="optional"
default="UNDEFINED" />
  ...
</xs:complexType>
```

Then, in the XML file for the `Component`, the attribute and its value may be specified as shown:

```
<LAMPWHEEL xmlns="urn:schemas-cosylab-com:LAMPWHEEL:1.0"
xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
LampWheelDescription="Example of a dumb lamp wheel"
>
...

```

⁶ In order for the schema file to be properly installed at build time (e.g. by using "make install"), the developer must edit the `CDB_SCHEMAS` variable of the Makefile. In our example, we have created `FILTERWHEEL.xsd` in `config/CDB/schemas` and we have added the following line in the Makefile:

```
CDB_SCHEMAS = FILTERWHEEL
```

⁷ Only the relevant part of the `xml` and `xsd` files are reported in this document. Refer to the ACS distribution for their complete version.

The value of this user-defined attribute is then read in the initialize method of the component (see source code in acsexmpl module for the complete code listing for C++) with the following code:

```

C++:

CDB::DAL_ptr dal_p = getContainerServices()->getCDB();
CDB::DAO_ptr dao_p = dal_p->get_DAO_Servant(m_fullName.c_str());
char* descr = dao_p->get_string("LampWheelDescription");

Java:

try {
    com.cosylab.CDB.DAL dal = getContainerServices().getCDB();
    com.cosylab.CDB.DAO dao = dal.get_DAO_Servant(m_fullName);
    String descr = dao.get_string("LampWheelDescription");
}
catch(alma.acs.container.ContainerException ex) {
    // unable to get DAL from the container
}
catch(com.cosylab.CDB.XMLerror ex) {
    // problem with the XML in the CDB
}
catch(com.cosylab.CDB.RecordDoesNotExist ex) {
    // no record by that name found in CDB
}

Python:

import Acspy.Util.ACSCorba
dal = Acspy.Util.ACSCorba.cdb()
dao = dal.get_DAO_Servant(m_fullName)
descr = dao.get_string("LampWheelDescription")

```

For more complex situations, in order for the DAO to be able to read your user-defined data, the XML and the XSD must be written following a mapping between the type of the data to be stored in the XML and the way it is written in the XML/XSD files; this mapping is explained in Section 4.2.1. You are free to create the XML and the XSD files without following the mapping, but in that case in order to read the data from the CDB you would have to get the raw XML (using the `get_DAO` method of the DAL) and parse the XML on your own, using whatever XML parser you prefer. This last situation is out of the scope of this document but you can look at the `acsexmplLampWheel` in the `acsexmpl` module, where the XML is parsed using the `expat` XML parser, to see how this can be done.

4.2.1 Mapping to XSD

In order to extend the XML to add user-defined data to a record of the configuration database, the XSD must be written according to the following mapping:

- **Classes:** Every class maps to an XSD complex type with the exact same name.
- **Interfaces:** Every interface maps to an XSD complex type with the exact same name.
- **Inheritance:** Single inheritance is implemented using the restriction or extension XML complex type inheritance. An intermediate type is introduced with an underscore prefix when both restriction and extension are required simultaneously. Multiple inheritance is achieved by containing instances of base classes. For example, if D is derived from A and B, element D will contain subelements A and B, of complex types A and B, respectively.
- **Simple attributes:** Class attributes which have a simple type are mapped into XML attributes, whose name is the same as the class attribute. Their types are mapped into the XSD equivalents (for example, strings map into xs:string).
- **Complex by-value attributes:** Class members which are of a complex type and are contained by value are mapped into XML sub-elements, whose name is the same as the class member's name, and whose type matches the complex type.
- **Complex by-reference attributes:** Class members which are of a complex type and are contained by reference are mapped into an element of the same name, which has only a single attribute, ref, of type xs:string. The value of this attribute may be used as a token with which the dereferencing can be made (e.g., a CURL).
- **Array attributes:** Array attributes are mapped into elements of the same name. The XML element contains one XML subelement per each element in the array. The XML subelements form a sequence. If the subelements are simple types, the XML subelement name is _, and it carries only one attribute whose name matches the name of the simple type. Otherwise, the name of the subelement is the name of the complex type.
- **Map attributes:** Map attributes are mapped into elements of the same name. The XML element contains one XML subelement per each element of the map. The subelement has at least one attribute, Name, which specifies its key. In other respects, maps are represented in the same way as arrays.

acsexmplFilterWheel is an example that shows how to use this mapping. In this example we have a filter wheel and we want to persistently store in the CDB the exact position of each slot as well as the type of each filter mounted. The XML is:

```

01 <FILTERWHEEL FilterWheelDescription="Example" AvailableSlots="6">
02     <position />
03     <desc />
04     <slots />
05     <Filter>
06         <_ Name="Red" Delta="140" Slot="0" />
07         <_ Name="Green" Delta="-346" Slot="3" />
08         <_ Name="Blue" Delta="12" Slot="1" />
09     </Filter>
10     <SlotStep>
11         <cdb:_ long="8123"/>
12         <cdb:_ long="15432"/>
13         <cdb:_ long="23698"/>
14         <cdb:_ long="53140"/>
15         <cdb:_ long="44325"/>
16     </SlotStep>
17 </FILTERWHEEL>

```

The XML shows the usage of the mapping. It contains

- an array of long describing the step, the position, of each slot of the wheel. The array, called SlotStep is described in lines 10 to 16: each sub element has a ‘_’ as tag with a single attribute with the name of the type of the stored value.
- a map of filter with the key represented by the name of the filter. The map, Filter, extends from row 5 to 9. Here you can see that the tag of each element of the map is ‘_’ and the key is represented by the attribute Name. Other attributes, Delta and Slot are also present.

The XSD (schema) for this example is:

```

01 <xs:schema ... >
02...
03 <xs:complexType name="FILTER">
04     <xs:attribute name="Name" type="xs:string" />
05     <xs:attribute name="Delta" type="xs:integer"/>
06     <xs:attribute name="Slot" type="xs:integer" />
07 </xs:complexType>
08
09 <xs:complexType name="FILTERWHEEL">
10 <xs:sequence>
11     <xs:element name="position" type="baci:ROdouble"/>
12     <xs:element name="desc" type="baci:ROstring"/>
13     <xs:element name="slots" type="baci:ROlong"/>
14
15     <xs:element name="Filter">
16         <xs:complexType>
17             <xs:sequence>
18                 <xs:element name="_" type="FILTER" maxOccurs="unbounded"/>
19             </xs:sequence>
20         </xs:complexType>
21 </xs:element>

```

```

22
23     <xs:element name="SlotStep" type="cdb:Array"/>
24
25 </xs:sequence>
26
27 <xs:attribute name="FilterWheelDescription" type="xs:string" use="optional"/>
28 <xs:attribute name="AvailableSlots" type="xs:integer" use="required" />
29 </xs:complexType>
30
31 <xs:element name="FILTERWHEEL" type="FILTERWHEEL" />
32</xs:schema>

```

The structure of the LAMPWHEEL tag is described from rows 9 to 29. An XML document adhering to the schema will contain only a FILTERWHEEL tag, as described in the schema on line 31.

The array SlotStep is defined in line 23. The attribute `cdb::Array` says that it is an array, but does not specify the type of the elements of the array. The type for the array elements is described in the XML, as previously mentioned.

The map, Filter, is described in lines 15-21. Line 18 states that the name of each tag is ‘_’ and the type is filter. Remember, the name must be ‘_’ as previously noted in the mapping description.

The type of each element of the map is described in lines 3-7. The key, Name, is in line 4 and it is another element required by the mapping.

The code to read the array of long (for C++) is also shown in the `acsexmplFilterWheel` module. After getting the DAL and the DAO, the following line reads the array of long named SlotStep, returning a CORBA sequence:

```

C++:
CDB::longSeq* lngSeq = dao_p->get_long_seq("SlotStep");

Java:
// within the context of a try-catch block (see other examples)
int longSeq[] = dao.get_long_seq("SlotStep");

Python:
longSeq = doa.get_long_seq("SlotStep")

```

To read the map is slightly more complicated:

C++:

```

01 CDB::stringSeq* fltKeys = dao_p->get_string_seq("Filter");
02 for (CORBA::ULong t=0; t < fltKeys->length() && t < availableSlots; t++)
03 {
04     char key[64];
05     char deltaName[128]; char slotName[128];
06     strcpy(key,((*fltKeys)[t]));
07     sprintf(deltaName,"Filter/%s/Delta",key);
08     sprintf(slotName,"Filter/%s/Slot",key);
09     int delta = dao_p->get_long(deltaName);
10     int slot = dao_p->get_long(slotName);
11     ... // Store the data
12 }

```

Java:

```

String filterString = "Filter/";
String deltaString = "/Delta";
String slotString = "/Slot";
try {
    DAO myDAO = m_dal.get_DAO_Servant("alma/FILTERWHEEL");
    String fltKeys[] = myDAO.get_string_seq("Filter");
    for(int t = 0; t < fltKeys.length && t < availableSlots; t++) {
        String key = fltKeys[t];
        String deltaName = filterString + key + deltaString;
        String slotName = filterString + key + slotString;
        int delta = myDAO.get_long(deltaName);
        int slot = myDAO.get_long(slotName);
    }
}
catch(com.cosylab.CDB.XMLerror ex) {
    // XML error
}
catch(com.cosylab.CDB.RecordDoesNotExist ex) {
    // no such record in CDB
}
catch(com.cosylab.CDB.FieldDoesNotExist ex) {
    // no such field in CDB
}
catch(com.cosylab.CDB.WrongDataType ex) {
    // incorrect data type
}

```

Python:

```

filterString = "Filter/"
deltaString = "/Delta"
slotString = "/Slot"
fltKeys = dao.get_string_seq("Filter")
for key in fltKeys:
    deltaName = filterString + key + deltaString
    slotName = filterString + key + slotString
    delta = dao.get_long(deltaName)
    slot = dao.get_long(slotName)

```

The first step is to read the keys(e.g. line 01 in the C++ example code) as an array of string. If you already know the keys you could optionally skip this step.

For each key, to read the Slot and the Delta we need to build the name (as shown in lines 7 and 8 for C++) using the key (e.g. line 6 for C++). Then the Slot and Delta attributes are read with a `get_long` (e.g. lines 9 and 10 for C++).

5 Modifying the data: WDAL and WDAO

The DAL and the DAO can only be used to *read* records or values from the CDB. If you need to *change* records in the CDB, you can do one of the following:

- manually edit the XML and/or XSD (using your preferred XML or text editor)
- use the `cdbBrowser` (see the `cdbBrowser` manual)
- use the WDAL and the WDAO from inside your program to modify the CDB programmatically

The WDAL is an IDL interface that extends the DAL with writing capabilities:

```
interface WDAL : DAL {
    WDAO get_WDAO_Servant(in string curl)
        raises(RecordDoesNotExist, RecordIsReadOnly, XMLerror);
    void add_node(in string curl, in string xml)
        raises(RecordAlreadyExists, XMLerror, CDBException);
    void remove_node(in string curl) raises(RecordDoesNotExist, RecordIsReadOnly);
    void set_DAO(in string curl, in string xml)
        raises(RecordDoesNotExist, FieldDoesNotExist, RecordIsReadOnly,
            XMLerror, CDBException);
};
```

It allows adding or removing a node, to replace an existing node with an xml string, as well as to get a WDAO remote object.

The WDAO is an IDL interface that extends the DAO with writing capabilities. It is similar to the DAO, but adds methods to *update* values (for each type of variable).

```

interface WDAO : DAO {
    void set_long(in string propertyName, in long value)
        raises(FieldDoesNotExist, FieldIsReadOnly);
    void set_double(in string propertyName, in double value)
        raises(FieldDoesNotExist, FieldIsReadOnly);
    void set_string(in string propertyName, in string value)
        raises(FieldDoesNotExist, FieldIsReadOnly);
    void set_field_data(in string propertyName, in string value)
        raises(WrongDataType, FieldDoesNotExist, FieldIsReadOnly);
    void set_string_seq(in string propertyName, in stringSeq value)
        raises(FieldDoesNotExist, FieldIsReadOnly);
    void set_long_seq(in string propertyName, in longSeq value)
        raises(FieldDoesNotExist, FieldIsReadOnly);
    void set_double_seq(in string propertyName, in doubleSeq value)
        raises(FieldDoesNotExist, FieldIsReadOnly);
};

```

Both the WDAL and the WDAO follow the mapping described previously. If you do not want to follow the mapping, then you must write the XML directly by using the WDAL `set_DAO` method, passing the XML directly as the second parameter.

`ascexmplFilterWheel` illustrates a C++ example of the writing capabilities offered by WDAL and WDAO. The following code updates the Filter map, changing the value of the attribute Delta of one entry.

```

C++:

01 CDB::DAL_ptr dal_p = getContainerServices()->getCDB();
02 CDB::WDAL_ptr wdal_p = CDB::WDAL::_narrow(dal_p);
03 CDB::WDAO_ptr wdao_p = wdal_p->get_WDAO_Servant(m_fullName.c_str());
04 ACE_CString deltaStr("Filter/");
05 deltaStr+=name;
06 deltaStr+="/Delta";
07 wdao_p->set_long(deltaStr.c_str(),delta);

Java:

DAL myDAL = null;
try {
    myDAL = m_containerServices.getCDB();
}
catch(alma.acs.container.ContainerException ex) {
    // unable to get DAL - abort/print out error here...
}

WDAL wdal = WDALHelper.narrow(myDAL);
try {
    // m_fullName is, e.g., "alma/FILTERWHEEL1"
    WDAO wdao = wdal.get_WDAO_Servant(m_fullName);

```

```

String deltaStr = "Filter/";
deltaStr += name;
deltaStr += "/Delta";
wdao.set_long(deltaStr, delta);
}
catch(com.cosylab.CDB.XMLerror ex) {
    // XML error
}
catch(com.cosylab.CDB.FieldIsReadOnly ex) {
    // requested field is read only
}
catch(com.cosylab.CDB.RecordIsReadOnly ex) {
    // requested record is read only
}
catch(com.cosylab.CDB.RecordDoesNotExist ex) {
    // requested record does not exist
}
catch(com.cosylab.CDB.FieldDoesNotExist ex) {
    // requested field does not exist
}
}

```

Python:

```

import Acspy.Util.ACSCorba
wdal = Acspy.Util.ACSCorba.cdb()
# m_fullName is, e.g., "alma/FILTERWHEEL1"
wdao = wdal.get_WDAO_Servant(m_fullName)
deltaStr = "Filter/"
deltaStr += name
deltaStr += "/Delta"
wdao.set_long(deltaStr, delta)

```

In Line 1 (C++ example code), a pointer to the DAL is obtained by calling the `getCDB` method of `ContainerServices`. In line 2, the `WDAL` is obtained via casting using the standard CORBA narrow technique. In line 3, the remote `WDAO` is obtained by calling the `get_WDAO_Servant` method; this is very similar to the `get_DAO_Servant` method call shown in Section 4.1.

To write the new Delta value, we first have to write the key (lines 4-6), similar to what was done when reading its value. In line 5, “name” is the key - for example “Red”.

The following code describes how to write the array of long, `SlotStep`:

C++:

```

01 CDB::longSeq* lngSeq =wdao_p->get_long_seq("SlotStep");
02 (*lngSeq)[(CORBA::ULong)slot] = step;
03 wdao_p->set_long_seq("SlotStep", *lngSeq);

```

Java:

```
01 int longSeq[] = wdao.get_long_seq("SlotStep");
02 longSeq[slot] = step;
03 wdao.set_long_seq("SlotStep", longSeq);
```

Python:

```
01 longSeq = wdao.get_long_seq("SlotStep")
02 longSeq[slot] = step
03 wdao.set_long_seq("SlotStep", longSeq)
```

In line 1 we read the CORBA sequence (i.e. the array of long), SlotStep. Then in line 2, we update the slot element of the CORBA sequence with the new value, step. Eventually, in line 3, we write the entire sequence to the CDB.

6 cdbjDAL

This command starts the Configuration Database service, the process name for which is cdbjDAL. The cdbjDAL process implements the IDL DAL (Data Access Layer) interface, which is the API for the ACS configuration database (CDB). The acsStartORBSRVC script (which is invoked automatically upon ACS startup via either the acsStart command or from within the ACS command center GUI) automatically executes it.

The Configuration Database service uses the following algorithm to resolve the path of the directory where the CDB XML files are located:

- Command line option `-root <path>`
- Environment variable `ACS_CDB`

Once the path has been resolved, it uses the directory to find the schemas and XML documents for the CDB.

When cdbjDAL is executed from within acsStartORBSRVC (which is, in turn, called from the acsStart script and/or when starting ACS from the command center GUI), it is not possible to pass the `-root` option. Therefore, in that case the only real usable way of defining the path for the CDB files is using the environment variable `ACS_CDB`. The default value for this variable provided by the login scripts is `$ACSDATA/defaultCDB`, where the ACS installation procedure puts the standard sample CDB.

At run-time, the DAL server searches for schema definitions, in the given order, in the following directories:

- \$ACS_CDB/CDB/schemas
- \$CWD/./config/CDB/schemas
- \$INTROOT/config/CDB/schemas
- \$ACSROOT/config/CDB/schemas

cdjDAL is a DAL server implemented as a JAVA CORBA servant. The main implementation class is `com.cosylab.cdb.jdal.Server`. cdjDAL can use both Jacorb or the JDK ORB. The command line argument `-jacorb` is passed down to the Java class by the cdjDAL startup script, since in ACS we normally want to use Jacorb. Other implementations of ORBs could be used as well, but might require slight changes due to the way that exporting the CORBA server is done (jDAL exports the CORBA server to listen at a specified port and name so the *corbaloc* from clients can be used - this is done differently depending upon the ORB vendor, so it should be checked if another ORB implementation is used). In the following table, other command line parameters are described:

Parameter name	Description
-root	The path of the directory where XML data is stored. jDAL searches data files relative to that directory. When a request is made for a record, jDAL will compose the full path to the XML data depending on this directory. The default is the current directory. Example: <code>-root \home\myData</code>
-o <filePath>	Specifies the file where jDAL ior should be written. If <i>filePath</i> is omitted, then <i>DAL.ior</i> file name in current directory is used.
-OApport	Specifies a TCP port different from the default
-orbacus	This parameter is here for backward compatibility because at the present ORBacus is no longer supported
-jacorb	If this parameter is present then Jacorb is used (for default it uses JDK ORB)

During its startup, jDAL initializes the XML parser factory using the JAXP standard. This means that it is possible to replace the XML parser with another JAXP compliant parser. By default, jDAL uses the Apache Xerces2 Java parser. Replacement with another JAXP parser is just a matter of ensuring that the *classpath* encounters a new implementation prior to the Xerces parser.

jDAL scans for all xsd (schema) files in root/schemas directory and in the directories mentioned above. Schema files found are added as external schema location files. The reason for this is the fact that jDAL ensures that all data in the system will be checked against its schema. In the case where a schema file is not found, an error message is printed to the console.

jDAL configures the XML parser to validate each XML file against its corresponding schema (xsd) file. If the XML parser doesn't understand the schema language, an error message is printed and XML CDB entries will be useless.

When a request comes to the jDAL server, it first checks to see if the requested XML entry exists. If there is no XML entry satisfying the request, a RecordDoesNotExist exception is raised. After the XML entry is found, the parsing is performed, validating against the schema file. In cases where a parsing error occurs, jDAL prints an error message and constructs/raises an XMLException so the client can see what is wrong with the data.

If everything is fine after the parsing phase, jDAL creates a DAO object or returns the expanded XML file to the requestor. Instantiated DAOs are transient objects and are destroyed by the POA when their reference counter drops to 0.

7 Command line utilities for dealing with the CDB

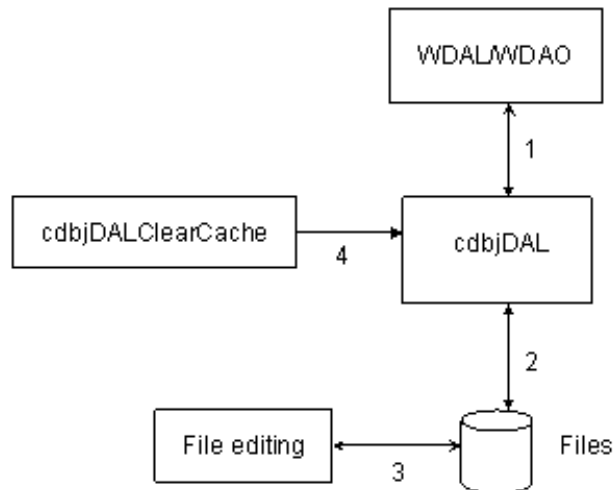
7.1 cdbjDALClearCache command

This command requests the Configuration Database (jDAL implementation) to clear its internal cache and notify applications (in particular the Manager) that updated data may be available. Using the cdbjDALClearCache command, it is easy to modify the CDB and make new values "active" without having to restart ACS processes. The procedure is as follows:

- Using a text editor or XML editor, edit the CDB XML files
- Issue the cdbjDALClearCache command to notify the CDB's DAL server (jDAL) that it should clear its cache
- The Manager automatically gets notified, and then updates and reloads its data from the CDB upon the next request

The cdbjDALClearCache command is very useful when the user changes the CDB contents and wants to notify the Manager of the new changes without restarting ACS and the services. The cdbjDALClearCache command forces the DAL server to invalidate its cache. For each subsequent new request for data that the DAL server, data is read directly from the configuration database rather than from the DAL's cache; this will prevent reading "stale" data from the cache.

If the CDB is edited through the cdbBrowser there is no need to issue a cdbjDALClearCache command because the cdbBrowser talks directly to jDAL through the WDAL/WDAO interfaces and the server will update the files after having updated its internal cache. The following figure explains this behavior:



When the user edits the CDB using the `cdbBrowser`, the changes are notified to the server (1) and then the server updates the files (2). If the user edits the files with a text editor (3) he has to notify the server about the changes he did by issuing a `cdbjDALClearCache` command (4). This command says to the server that the cache it has in memory disagrees with the files on the file system and has to be refreshed when the data are accessed again.

7.2 `cdbjDALShutdown` command

This command shuts down the Configuration Database service. The `acsStopORBSRVC` script, which is invoked from the `acsStop` script (or when stopping ACS from the `acscommandcenter` GUI) executes this command, so the user will normally not need to run this command.

It is possible to specify the host and port where the `cdbjDAL` is running in the command line:
`-k corbaloc::`

7.3 `cdbRead` command

This command is a utility to dump DAOs to the standard output.

The calling syntax is:

```
cdbRead <DAO name> [-raw]
```

where `<DAO name>` is the hierarchical name of the DAO in the CDB, for example `cdbRead /alma/LAMP1`. The command produces output such as the following:

```
Node brightness
  description="brightness"
  units="%"
  min_step="1.0"
```

```

min_value="0.0"
max_value="100"
default_timer_trig="10000000"
min_timer_trig="10000"
min_delta_trig="0"
default_value="0.0"
graph_min="-1.7976931348623157E+308"
graph_max="1.7976931348623157E+308"
archive_delta="0"
format="%9.4f"
resolution="65535"
archive_priority="3"
archive_min_int="0"
archive_max_int="0"

```

The `-raw` parameter after the DAO name, dumps the DAO as a raw XML file.

It is possible to specify the host and port where the `cdjDAL` is running in the command line:

```
-k corbaloc::

```

7.4 CDB Browser

This graphical application can be used to browse and edit the run-time configuration database [RD07]. The `cdjBrowser` command line accept the `-k` parameter:

```
-k corbaloc::

```

7.5 CDB Checker

This application can be used to check a CDB for consistency and correctness. A separate document is being prepared to explain its proper usage; explaining its proper usage is therefore outside the scope of this document.

The synopsis of the `cdjChecker` command is:

```
cdjChecker [-v] [path to CDB XML files] [path to CDB XSD files]
```

All xml files recursively found in [path to CDB XML files] are checked. The optional argument contains a ":" separated list of files and directories with absolute path. Normally this command line argument points to the root directory of the CDB to be checked, i.e. it is `$ACS_CDB/CDB`. If not [path to CDB XML files] is given, `$ACS_CDB` is used by default.

All xsd files are recursively found in [path to CDB XSD files] are checked in addition to the standard search path automatically build and passed in the `ACS.cdbPath` property. The optional argument contains a ":" separated list of files and directories with absolute path.

The other parameters that can be specified in the command line are:

-v	Verbose: this option provides details about the checking process
-n	Download schemas from the internet

-r	Disable the recursively search of .xsd and .xml files

8 Resolving the Configuration Database Reference

ACS applications (like `acsStartManager` or `acsStartContainer`) use the following algorithm (in the order listed) to resolve the DAL reference:

- Command line option `-d` or `-DALReference`
- Environment variable `DAL_REFERENCE`
- Using generated reference: `corbaloc::<hostname>:<dal_port>/DAL`

9 Manager Configuration Database

The Configuration Database for the Manager is in the database branch: `/MACI/Managers/Manager`. The definition of the configuration parameters for the Manager is in its schema file: `$ACSRROOT/config/CDB/schemas/Manager.xsd`⁸.

Important parameters are (for a complete list, consult the schema file):

- **CommandLine:** Default command-line added to given command-line
- **Startup:** List of Components to be automatically started on startup
- **CacheSize:** number of logs to be cached before logging
- **MinCachePriority:** minimum log priority (messages with lower priority are ignored)
- **MaxCachePriority:** maximum log cache priority (messages with higher priority are not cached and are logged immediately)

For each Component in the system, the Manager must be able to find, on request, all information needed to return its reference to clients and to start/stop it when commanded. Therefore, the Configuration Database used by the Manager must contain the mandatory configuration file: `/MACI/Components/Components.xml`.

The definition of the Components configuration file is in the schema file: `$ACSRROOT/config/CDB/schemas/Components.xsd` and consists of an array with one entry per each known Component, as in the following example:

```
<_ Name="PBEND_B_01" Code="acsexmplPS" Type="IDL:ALMA/PS/PowerSupply:1.0"
Container="Container"/>
```

⁸ See the schema documentation for further details.

The attributes of the array have the following meaning:

- **Name:** the name of the component
- **Code:** the code of the component (in UNIX it is typically a shared library name for C++). Changing the value of this attribute allows running different implementations of the same IDL interface.
- **Type:** the IDL interface implemented by the component
- **Container:** the container where the component is to be deployed. Changing the value of this attribute allows relocating the component.

Changing the *Code* and/or the *Container* attribute of a component's entry allows one to run different implementations of a single IDL interface and/or (re)deploy a component from one container to another. This operation can be done without restarting ACS in this way:

- stop the component in such a way that it is unloaded
- change the Code and/or Container attribute of the entry of that Component in Components.xml.
- clear the cache of the CDB with `cdbjDALClearCache`
- restart the component

10 Container Configuration Database

The Configuration Database for a Container is in the database branch:
/MACI/Containers/<Container name>.

The CDB entry for a C++ Container is optional (defaults are used if not present), while Java Containers currently do not support storing configuration parameters in the CDB. Python Containers have only limited support for Container info at the present.

The definition of the configuration parameters for the C++ Container is in schema file:
\$ACSR00T/config/CDB/schemas/Container.xsd

Important parameters are (consult the schema file for the complete list):

- **CommandLine:** Default command-line added to given command-line
- **ManagerReference:** The Manager reference in the format
corbaloc::<host>:<port>/Manager

- **Autoload:** DLLs to be loaded automatically on Container startup
- **CacheSize:** Number of logs to be cached before logging
- **MinCachePriority:** Minimum log priority (messages with lower priority are ignored)
- **MaxCachePriority:** Maximum log cache priority (messages with higher priority are not cached and are logged immediately)

Whenever the Manager requests a Container for a (C++) Component, it passes to it information about the DLL to be loaded. Similarly, Java and Python Components require information in the CDB to allow the Manager to activate them.

11 Component Configuration Database

For simple ACSComponents, you only need to put static deployment information in the configuration database; you have to supply some information which the Manager needs for all statically defined instances of your Component. For example, you should fill in the file:

CDB/MACI/Components/Components.xml

adding entries for the instances you want to statically define.

12 Characteristic Component Configuration Database

Characteristic Components, i.e. Components implemented according the BACI Design Patterns [RD06], keep the configuration for their Properties and Characteristics in the CDB. Each Characteristic Component looks for its configuration information in the database branch: `/alma/<Component name>`. The actual structure of the database depends on the type of Component, but will essentially contain characteristics for each Property as defined in `$ACSROOT/config/CDB/schemas/BACI.xsd`. You can find a detailed description of the format of the XML file for a CharacteristicComponent in [RD06].

As we said before, the developer can customize the XML file describing a component by changing its schema file (see Section 4.2).

13 Alternative CDB structure for component's deployment

Instead of just having one file, Components.xml, in your CDB you have the option to replace it with a separate directory which includes a single xml file for each Component. To support this structure, ACS has added an additional schema, Component.xsd (notice that the schema name is singular), to be used instead of the Components.xsd schema that the Components.xml file uses.

For example, if you previously had \$ACS_CDB/CDB/MACI/Components/Components.xml Where the Components.xml file looked as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

  <_ Name="MOUNT1"      Code="acsexmplMountImpl"
                        Type="IDL:alma/MOUNT_ACS/Mount:1.0"
                        Container="bilboContainer"/>
  <_ Name="MOUNT2"      Code="acsexmplMountImpl"
                        Type="IDL:alma/MOUNT_ACS/Mount:1.0"
                        Container="bilboContainer"/>
  <_ Name="HELLODEMO1"  Code="alma.demo.HelloDemoImpl.HelloDemoHelper"
                        Type="IDL:alma/demo/HelloDemo:1.0"
                        Container="frodoContainer"/>
  <_ Name="HELLOLAMP1"  Code="alma.demo.LampAccessImpl.LampAccessHelper"
                        Type="IDL:alma/demo/LampAccess:1.0"
                        Container="frodoContainer"/>
</Components>
```

The new CDB structure you would have would look like:

```
$ACS_CDB/CDB/MACI/Components/MOUNT1/MOUNT1.xml
$ACS_CDB/CDB/MACI/Components/MOUNT2/MOUNT2.xml
$ACS_CDB/CDB/MACI/Components/HELLO1/HELLO1.xml
$ACS_CDB/CDB/MACI/Components/HELLO2/HELLO2.xml
```

An example of the xml file (C++ example for the MOUNT1.xml) would be

```
<?xml version="1.0" encoding="utf-8"?>
<Component xmlns="urn:schemas-cosylab-com:Component:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  Name="MOUNT1"
  Code="acsexmplMountImpl"
  Type="IDL:alma/MOUNT_ACS/Mount:1.0"
  Container="bilboContainer"
/>
```

(Java example for HELLO1.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<Component xmlns="urn:schemas-cosylab-com:Component:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  Name="HELLODEMO1"
  Code="alma.demo.HelloDemoImpl.HelloDemoHelper"
  Type="IDL:alma/demo/HelloDemo:1.0"
  Container="frodoContainer"
/>
```

About this CDB structure, there are important things to note:

- The Component's xml filename must match the directory name in which it resides. For example, if you create \$ACS_CDB/CDB/MACI/Components/COMP1, the xml file must be called COMP1.xml
- Some Components, particularly C++ Components, require instance xml files. This is usually associated with Components which use properties (i.e. CharacteristicComponents). In this case the file hierarchy in the \$ACS_CDB/CDB/MACI/Components directory and the \$ACS_CDB/alma directory must match. For example if you have \$ACS_CDB/CDB/MACI/Components/MOUNT1/MOUNT1.xml then you must have \$ACS_CDB/CDB/alma/MOUNT1/MOUNT1.xml

If you are not using properties in C++ you do not need this matching because you do not have an entry in \$ACS_CDB/CDB/alma. Likewise for Java components or Python components, which usually do not require properties (support for CharacteristicComponents in Java and Python is still not entirely complete), you can call the new directory and xml file anything you want.

14 Hierarchical components and CDB structure

In a small system, with few Components, each Component can be given a simple name and they can just be deployed using this name by writing entries in the MACI/Components/Components.xml file. But for a more complex system, this file becomes very big and it may become difficult to see the relationships between the various Components.

Most systems have a naturally hierarchical logical structure, i.e. it is possible to group Components according to functional groups and sub-groups and with logical and/or physical (for hardware devices) containment rules. The ACS CDB allows you to configure Components according to these rules.

The following is an example of a hierarchical structure for the Components of the system, where the names in **BOLD** correspond to actual Components, while the other names are simply logical grouping names and do not correspond to actual Components.

```

ALMA_DOOR
ALMA_BACK_DOOR
  LAMP
CONTROL
  ANTENNAS
    ANTENNA_1
      MOUNT
        LAMP_1
        POWER_SUPPLY
    ANTENNA_2
      MOUNT
        POWER_SUPPLY
    ANTENNA_3
      MOUNT
        POWER_SUPPLY
    ANTENNA_H
      MOUNT
        POWER_SUPPLY
TOWER_1
  FRONTDOOR
TOWER_H
  FRONTDOOR
TestInclude_01
TestInclude_02
TestInclude_03
... dynamic components defined using XInclude ...

```

From the logical point of view, the complete name of each Component is built taking the hierarchical path using the '/' path separator.

For example:

- CONTROL/ANTENNAS/ANTENNA_1
Is a logical entity to group all Components that have to do with ANTENNA_1.
- CONTROL/ANTENNAS/ANTENNA_1/MOUNT
Is the actual Component for the Mount of ANTENNA_1
- CONTROL/ANTENNAS/ANTENNA_1/MOUNT/LAMP_1
Is another Component logically in ANTENNA_1
Moreover this LAMP Component is also part of MOUNT
Remember that this is just a logical containment and not a physical containment.
CONTROL/ANTENNAS/ANTENNA_1/MOUNT **and**

CONTROL/ANTENNAS/ANTENNA_1/MOUNT/LAMP_1 might be physically deployed in different Containers and in different hosts⁹

The configuration of the deployment of the system is done in the branch MACI/Components of the configuration database.

14.1 Components.xml

The simplest option consists in listing all Components in the Components.xml file according to the schema urn:schemas-cosylab-com:Components:1.0. Hierarchical names can be put here by putting the complete name with '/' separators in the Name attribute of the Component.

For our example we have simply put here two Components that are at the root level of the hierarchy:

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xi="http://www.w3.org/2003/XInclude">
  <_ Name="ALMA_DOOR"
    Code="acsexmplDoorImpl"
    Type="IDL:alma/acsexmplBuilding/Door:1.0"
    Container="Container" />
  <_ Name="ALMA_BACK_DOOR"
    Code="acsexmplDoorImpl"
    Type="IDL:alma/acsexmplBuilding/Door:1.0"
    Container="Container" />
</Components>
```

14.2 Hierarchical components as directory tree

Another option to build hierarchies consists of creating layers:

- one directory per layer with the name of the directory matching the name of the Component
- one XML file in each directory named <Component-Name>.xml

TOWER_1 is such an example:

A TOWER

- has a main Component
- and contains a FRONTDOOR sub-Component

Therefore we have:

⁹ r example because the LAMP_1 is controlled by an analog I/O board in another computer.

Usually this logical hierarchy means that LAMP_1 is essential for the functioning of the parent node MOUNT and activated by it. But this is not a requirement.

- TOWER_1 directory
 - TOWER_1.xml file, following the urn:schemas-cosylab-com:Component:1.0 schema to describe one single Component
 - FRONTDOOR directory
 - ✦ FRONTDOOR.xml file, following the urn:schemas-cosylab-com:Component:1.0 schema to describe one single Component

With this solution the xml file in each directory describes just a single Component. On the one hand, this structure makes it very easy to add and remove Components. On the other hand, it is more difficult to get an overview of the entire CDB structure from the UNIX command line compared to using a single Components.xml file. The cdbBrowser, a good editor, and/or the usage of more advanced UNIX commands may be used, however.

14.3 Hierarchical components as single file

The second option is to group an entire sub-hierarchy into one single file.

TOWER_H is such an example:

- It has a TOWER similar to the example above.
- But in this case, everything is described in the TOWER_H.xml file, following the urn:schemas-cosylab-com:HierarchicalComponent:1.0 schema to describe the root component and the FRONTDOOR sub-component:

```
<?xml version="1.0" encoding="utf-8"?>
<HierarchicalComponent xmlns="urn:schemas-cosylab-com:HierarchicalComponent:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  Name="TOWER_H"
  Code="acsexmplBuildingImpl"
  Type="IDL:alma/acsexmplBuilding/Building:1.0"
  Container="Container" >
  <_ Name="FRONTDOOR"
    Code="acsexmplDoorImpl"
    Type="IDL:alma/acsexmplBuilding/Door:1.0"
    Container="Container"/>
</HierarchicalComponent>
```

This structure gives an advantage when we are dealing with true hierarchical Components that always must be deployed together. In this way, a single file is sufficient to describe the deployment of the whole hierarchy.

14.4 Deep hierarchy with pure-logical nodes

In many cases it is useful to group Components under logical nodes that do not correspond to actual Components.

For example we want to group all Components in the Control System under CONTROL, but CONTROL is not a Component itself.

In this case it is sufficient to create a CONTROL directory without putting a Control.xml file and then sub-directories for the child layers.

The CONTROL/ANTENNAS/... hierarchy is an example of this.

14.5 Hierarchy with pure-logical nodes and sub-nodes in one file

One might also want to put multiple sub-nodes inside the same logical node. This is a parallel to the previous case where we have put the definition of both the TOWER_H component and of its sub-component(s) FRONTDOOR in a single file.

Let's assume we want to do a similar thing for CONTROL/ANTENNAS/ANTENNA_H and put the entire ANTENNA definition in a single file.

Since ANTENNA_H is just a logical node and does not correspond to a Component, we cannot use the HierarchicalComponent schema as above. But we can use the Components schema as for the Components.xml file. We create then the file

CONTROL/ANTENNAS/ANTENNA_H/ANTENNA_H.xml like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xi="http://www.w3.org/2003/XInclude">
  <_ Name="MOUNT"
    Code="acsexmplMountImpl"
    Type="IDL:alma/MOUNT_ACS/Mount:1.0"
    Container="Container"/>
  <_ Name="POWER_SUPPLY"
    Code="acsexmplPowerSupplyImpl"
    Type="IDL:alma/PS/PowerSupply:1.0"
    Container="Container" />
</Components>
```

In principle, the whole CONTROL configuration could be placed in just one single CONTROL.xml file, but this is not advisable because then we would lose the modularity of the configuration database.

14.6 Putting multiple XML files in the same directory using XInclude

In all these examples we always have just one single xml file per directory in the hierarchy.

It is also possible to put multiple xml files in the same directory, each with a number of components inside, by using the recent **XInclude** and **XPointer** XML specification.

- We can put Component specifications in separate files to be included in a main Components.xml

- Each of these external files must be a well formed Components description file
- The Components described in such files are included in the main `Components.xml` using the following syntax:

```
<xi:include href="<relative path>/MyIncludeFile.xml" xpointer="element(/1)" />
```

Using this technique include XML files can be in principle generally used everywhere in the CDB.

The only hard constraint is that:

- **the name of the include file cannot be the <directory>.xml,** because this is the file that the CDB engine will try to load directly when navigating the CDB structure.

We can put in the Components directory (or in a subsystem sub-directory) the file `IncludeTest.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <_ Name="TestInclude_01"
    Code="acsexmplPowerSupplyImpl"
    Type="IDL:alma/PS/PowerSupply:1.0"
    Container="Container"/>
  <_ Name="TestInclude_02"
    Code="acsexmplPowerSupplyImpl"
    Type="IDL:alma/PS/PowerSupply:1.0"
    Container="bilboContainer" />
  <_ Name="TestInclude_03"
    Code="acsexmplPowerSupplyImpl"
    Type="IDL:alma/PS/PowerSupply:1.0"
    Container="bilboContainer" />
</Components>
```

- Using the hierarchical CDB structure describe here above, CORRELATOR might have the file `CORRELATOR/IncludeComponents.xml` inside the CORRELATOR directory.
- Both files will be included in the main `Components.xml` eventually together with other component specifications:

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xi="http://www.w3.org/2001/XInclude">
  <_ Name="ALMA_DOOR"
    Code="acsexmplDoorImpl"
    Type="IDL:alma/acsexmplBuilding/Door:1.0"
    Container="Container" />
  <xi:include href="IncludeTest.xml" xpointer="element(/1)" />
  <xi:include href="CORRELATOR/IncludeComponents.xml" xpointer="element(/1)" />
</Components>
```

This technique is particularly useful to define Dynamic Components in multiple files per each subsystem.

14.7 Deployment of Dynamic Components in multiple files.

DynamicComponents are described in the `Component.xml` file by entries whose name is `"*"`.

This works fine if we are happy to put all DynamicComponents in the same `Components.xml` file. But it is impossible to place their description in a separate, own file, because:

- We would have to create a directory called `"*"` containind a file calles `*.xml`
- We would be in any case limited to just one component.

This is a problem when trying to factorize the CDB in a structured and hierarchical way.

It is possible instead to use the recent **XInclude** and **XPointer** XML specification.

Using the hierarchical CDB structure describe above , `CORRELATOR` might have the file `CORRELATOR/IncludeDynamic.xml` inside the `CORRELATOR` directory:

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <_ Name="*"
    Code="corrDynamic_1"
    Type="IDL:alma/MountDynamic_1:1.0"
    Container="Container" />
  <_ Name="*"
    Code="corrDynamic_2"
    Type="IDL:alma/MountDynamic_2:1.0"
    Container="corrContainer" />
</Components>
```

The file will be included in the main `Components.xml` eventually together with other component specifications:

```
<?xml version="1.0" encoding="utf-8"?>
<Components xmlns="urn:schemas-cosylab-com:Components:1.0"
  xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
  xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xi="http://www.w3.org/2001/XInclude">
  <_ Name="ALMA_DOOR"
    Code="acsexmplDoorImpl"
    Type="IDL:alma/acsexmplBuilding/Door:1.0"
    Container="Container" />
    <xi:include href="CORRELATOR/IncludeDynamic.xml" xpointer="element(/1)"
  />
</Components>
```

Appendix A. Obsolete CDB Table interfaces

The CDB Table interfaces are now obsolete and should be replaced by the DAL interfaces.

Since the CDB Table is still being used internally and by some user application, we keep here in appendix the design documentation.

We will remove this appendix when there is no longer code using them.

At the heart of this interfaces to access a Configuration Database, is an abstract interface named 'Table' and a simple pattern for registering, creating, accessing, and destroying object instances that implement interfaces.

The core pattern is a singleton class called TableStorage. Its functionality is accessed through a small set of exported functions.

Every implementation must be registered in the CDB by providing its factory functions.

Currently there are four different implementations of the Table interface and the CDB module itself registers them. Those implementations are named INI, IMDB, CCS and DAL.

- INI - data is stored in plain text file.
- IMDB - data is stored in memory only
- CCS - data is stored in the VLT CCS database
- DAL - data is based in XML files and accessed through a remote ACS DAL server and could be stored depending on the DAL server implementation. This implementation allows to use the Table interfaces to access the ACS XML based Configuration Database.

These predefined interfaces and the mechanism for registering/instantiating implementations provides a very flexible way for using different methods of data storage. Even other

implementations can be mixed and coexist at runtime, completely transparent to the clients. For example, one Component can use INI while another Component uses CCS, and both of them can coexist in a container that itself uses a DAL implementation. Components use the CDB method `getDatabase()` without worrying about its type, but they can require special type and configuration if needed.

CDB is a form of shared library and therefore can be used without tight coupling to other ACS modules. New implementations for the CDB must inherit from the Table interface and provide its virtual functions. After an interface is implemented, a simple registration function must be called to register the new implementation to the CDB. The registration function has the following prototype:

```
void cdb::registerTable( const char* name, TableFactory pTf );
```

Parameter *name* is the name of new implementation while *pTf* is a factory function. This function is called from CDB every time a new instance of implementation is required.

The factory function is pretty simple and its declaration is:

```
Table* createTable( int argc, char** argv, CORBA::ORB_ptr orb );
```

For example, a factory function for a CCS table looks like this:

```
Table* CCS::createTable( int argc, char** argv, CORBA::ORB_ptr orb)
{
    const char* pProcess = "";
    if( argc > 0 )
        pProcess = argv[0];
    return new CCS( pProcess );
}
```

The CDB module calls this registered function in standard way - using `argc`, `argv`. All entries given by the user at the command prompt are passed to the factory function so that the implementation is free to interpret command line arguments. In the example given above, the factory function for the CCS tables uses `argv[0]` to create a new instance of CCS implementation since the CCS implementation requires that information. On the other hand, the DAL factory function simply passes given parameters to the constructor of DAL table, which scans for keywords in order to configure itself. In such a way, any implementation is free to decide what parameters to use and how the parameters will be interpreted. An additional parameter in the factory function prototype is the ORB pointer - *orb*. This parameter is added for implementations of the Table interface that want to reuse an ORB instance created by other modules (i.e., Activator). Such an implementation is the DAL. The ORB can be resolved by using `argc/argv` too, and therefore the third parameter is redundant but it is added to make it more visible to the developers of new tables. The CDB itself uses one command line parameter named **ACS_CDB**.

This parameter is used to choose which type of registered instances should be used as default. For example the command line:

```
maciContainer -ACS_CDB CCS
```

tells CDB to use the CCS implementation as default.

To access an instance of database access, a simple function is exported:

```
Table* getDatabase( int argc = 0, char** argv = NULL, CORBA::ORB_ptr orb =  
CORBA::ORB::_nil(), const char* defaultTable= NULL, int forceNew = 0 );
```

This function is written in a such way that the programmer simply invokes `getDatabase()` when he needs a DAO and forgets about it after that. In other places, the same approach can be used without worrying about creation, destruction lifetime, and memory leaks of such a constructed Table object. The TableStorage class will take care of creation of that object and will destroy it if the programmer forgot to.

The Table interface was chosen because of simplicity for the first integration of the new CDB with existing modules. The new CDB interface is hidden behind the Table interface and because of that, existing code can use the new CDB without any modifications.