CUDB LDAP Data Views Management

Advanced

USER GUIDE

11/1553-HDA 104 03/10 Uen A

Contents

1 Introduction

This document describes the concepts, configurations, and procedures of the Lightweight Directory Access Protocol (LDAP) Data Views function in the Ericsson Centralized User Database (CUDB) system.

In case the LDAP Data Views function is available, CUDB supports accessing stored data through customizable views.

Note: The LDAP Data Views function can only be used if the Application Facilitator Value Package is available.

1.1 Target Groups

The intended audience of this document is the same audience as of the *CUDB System Administrator Guide*, Reference [1]: Ericsson personnel and operators. A general knowledge of the CUDB system is assumed.

1.2 Revision Information

Rev. A

This document is based on 11/1553-HDA 104 03/9 with the following changes:

- Section 1 on page 1 and Section 3.2 on page 21: Updated with information on function availability, regarding the Application Facilitator Value Package.
- Section 3.2.1 on page 22, Section 3.2.2 on page 22, Section 3.2.3 on page 23, and Section 3.2.4 on page 23: Updated procedure information.

1.3 Typographic Conventions

Typographic conventions can be found in the following document:

- *Typographic Conventions*

2 Overview

The CUDB LDAP Data Views function is based on assigning views to LDAP users. If the LDAP user sending a request has a configuration assigned LDAP data view, then that user can access the data through that view. The handled entries, attributes, and object classes belong to that view.

The main principle of the LDAP Data Views function is to present the existing data stored in the core DIT in CUDB in a different way: through to a custom LDAP tree, and also using custom object classes and attributes.

Refer to the *Concepts* section of *CUDB LDAP Data Views*, Reference [2] for more information on the basic concepts of LDAP Data Views.

Note: To follow this document, a general knowledge of the LDAP Data Views function is necessary, refer to *CUDB LDAP Data Views*, Reference [2] for more information.

3 LDAP Data Views Configuration

LDAP Data Views can be configured on a CUDB node if several actions (mandatory and optional) are performed. Data views export the core DIT in a different way and if some application specific data is missing in the core DIT, a manual addition is necessary. A schema update is required to define new object classes and attributes, and where this data is to be stored. For more information about schema update, refer to *CUDB Node Schema Update*, Reference [3]. The new data is then added –typically through provisioning– into the already existing or new entries in the core DIT.

After the core DIT has all the necessary data, the next step is to define the LDAP Data View virtual tree (for more information, refer to the *How to Use This Function* section of *CUDB LDAP Data Views*, Reference [2]). The virtual tree is built from the root down, following the structure expected by the application Front End (FE), defining the Distinguished Names (DNs) of every virtual entry in this virtual tree. For every virtual entry in the virtual LDAP tree, it is necessary to define which object classes must (STRUCTURAL) or may (AUXILIARY) be included in it. Also, for every object class in every virtual entry in the virtual tree, it is necessary to define which attributes should be present and from where in the core DIT (which entry and attribute) to retrieve their values. This definition is done by means of mapping rules inside the mapping configuration eXtensible Markup Language (XML) file. Object classes and attributes used in the mapping file are defined in the LDAP schemas.

Refer to the *Concepts* section of *CUDB LDAP Data Views*, Reference [2] for more information on the basic concepts of LDAP Data Views.

Each data view must have exactly one data view LDAP schema file named after the view, which is allowed to be empty. In this case, the virtual object classes and virtual attributes used in the mapping are supposed to be references to the already defined object classes and attributes of some schemas in the CUDB system. If different attribute names or different object classes from the ones in the core DIT are needed, they must be defined in the data view schema. Newly defined virtual attributes must already exist in the core DIT, with the names being the only differences. A virtual attribute must be of the same type as the real attribute in the association, since this is the only way for a virtual attribute to handle a value from the real attribute with a certain type.

Mandatory files for the LDAP Data Views configuration, the data view schema and the mapping configuration file must be placed in the `/home/cudb/dataAccess/ldapAccess/ldapFe/config/views/` directory on the CUDB node and must be named after the view. The view name is used during the creation of the data view in the data model, described in Section 3.2.1 on page 22. For more information on the data model, refer to *CUDB Node Configuration Data Model Description*, Reference [4].

The schema file must have the `.schema`, the mapping file must have the `.xml` extension.

3.1 Data View Configuration Entities

The LDAP Data Views tree is specified in the mapping configuration file as an XML structure following the allowed rules, described in more details later in this document. The mapping file starts with the `<view>` tag containing the hierarchical (nested) structure of virtual entries building up the tree. A virtual entry, like any LDAP entry, consists of object classes and attributes, and this structure must be specified in the mapping file. For each of these virtual entries, it is necessary to state the included object classes (structural, and optionally auxiliary). For all MUST attributes (and possibly MAY attributes) specified in the virtual entry object classes, a mapping rule is necessary to state from where in the core DIT the virtual entry attribute value is retrieved.

The general structure of a mapping is shown in Figure 1.

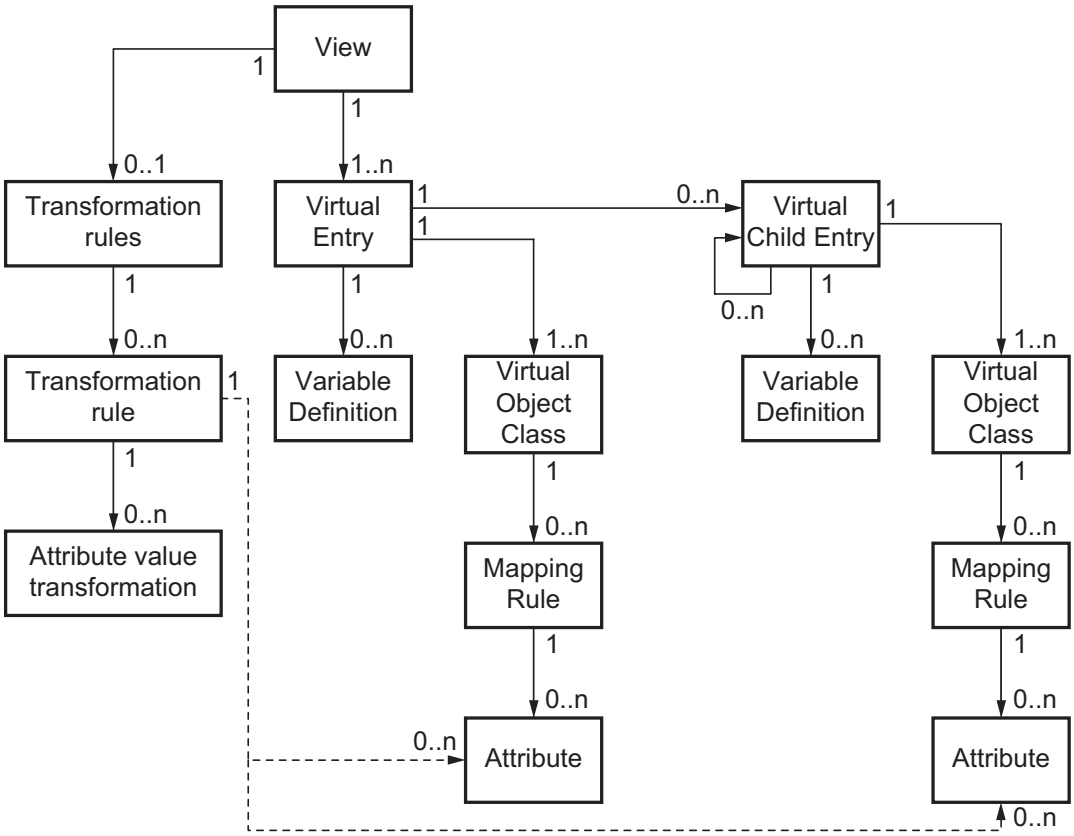

Figure 1 General Structure of a Mapping

See Example 1 and Figure 2 for an example of this general structure of mapping.

Each mapping file begins and ends with the tags <view>...</view>. These tags must contain the viewName attribute.

```
<view viewName="AppView">
  <virtualEntry dn="a=1">
    <virtualObjectClass name="oc_1">
     ..
    </virtualObjectClass>
    <virtualObjectClass name="oc_2">
      <mapRule dn="serv=EPS,mscId=10,ou=multiSCs,
          dc=example,dc=com">   <!-- MapRule 1 -->
            <attribute attrName="EpsProfileId" objectClass="EpsStaticInf" />
            <attribute attrName="SmMsisdn" objectClass="SmSessInf" newName="Sm-v" />
            ..
      </mapRule>
        ..
      <mapRule dn="serv=WLAN,mscId=10,ou=multiSCs,dc=example,dc=com" allAttributes="true">   <!-- MapRule n --
      </mapRule>
    </virtualObjectClass>
      ..
    <virtualObjectClass name="oc_n">
    </virtualObjectClass>

    <virtualEntry rdn="b=2">
          ..
        <virtualEntry rdn="c=3">
          ..
        </virtualEntry>
    </virtualEntry>
    <virtualEntry rdn="d=4">
      ..
    </virtualEntry>
  </virtualEntry>
</view>
```

Example 1 Mapping Structure Example

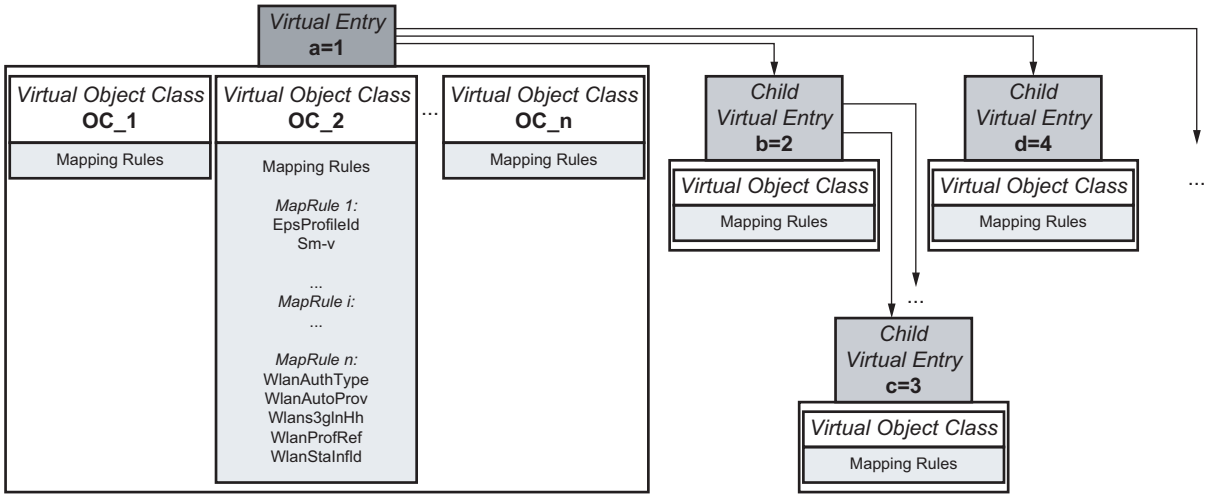

Figure 2 Mapping Example

3.1.1 Virtual Entries

Virtual entries are fictive entries visible only from a view. These fictive entries appear only after accessed by an LDAP operation and all their mandatory elements from the mapping structure have the proper values. Virtual entries

are defined with the tags <virtualEntry>...</virtualEntry>. These tags must be nested between <view>...</view> and can be repeated as many times ([0..N]) as many virtual entries needed.

The first virtual entry specified in mapping is the root entry.

To define virtual entries at lower virtual tree levels, virtual entries can be nested into each other. For more information, see Section 3.1.2 on page 8.

The name of a first level virtual entry must be defined in the tag attribute dn and must be a full DN. Only the top level virtual entry is specified by the dn tag and has a full DN (such as dc=example,dc=com).

The naming attribute can be chosen arbitrarily, as long as it is any attribute in the core DIT schemas or in the data view schema.

Virtual entries are defined from top to the bottom of the tree.

An example for defining virtual entries is shown in Example 2.

```
<view viewName="AppView">
  <virtualEntry dn="dc=example,dc=com">
      <virtualObjectClass name="oc_1">
        ..
      <virtualObjectClass/>
  </virtualEntry>
</view>
```

Example 2 Virtual Entry Example

3.1.2 Child Virtual Entries

Child virtual entries are the same sort of virtual entries as the ones described in Section 3.1.1 on page 7, but they can be defined under the first or at a lower virtual level in the LDAP view tree. Child virtual entries are defined with the tags <virtualEntry>...</virtualEntry>. These tags must be nested between <virtualEntry>...</virtualEntry> and can be repeated as many times ([0..N]) as needed below the parent virtual entry.

The name of a child virtual entry must be defined in the tag attribute rdn.

Note: This tag attribute is different from the tag attribute of a virtual entry at the first virtual level. The value must be a Relative Distinguished Name (RDN) containing the virtual naming attribute of the (current) child virtual entry only (such as <virtualEntry rdn="a">).

The naming attribute can be chosen arbitrarily, as long as it is any attribute in the core DIT schemas or in the data view schema. A child virtual entry can be nested between the opening and closing tags of a current virtual entry. This must be done in the same way as nesting a child virtual entry between the tags of a virtual entry at the first virtual level. The nesting of child virtual entries can be repeated (recursively) as many times ([0..N]) as needed in the LDAP tree of the view.

Note: The method above is the only allowed way to define child entries. It is not permitted to use a full DN or an RDN with defining more than one virtual level by using more than one naming attribute at the same time.

An example for defining virtual child entries is shown in Example 3 and Figure 3.

```
<view viewName="AppView">
 <virtualEntry dn="dc=example,dc=com">
   ..
   <virtualEntry rdn="cn=aaa">
     ..
     <virtualEntry rdn="a=1">
       ..
       <virtualEntry rdn="c=3">
         ..
       </virtualEntry>
     </virtualEntry>
     <virtualEntry rdn="d=4">
       ..
     </virtualEntry>
   </virtualEntry>
 </virtualEntry>
</view>
```

Example 3 Virtual Child Entry Example

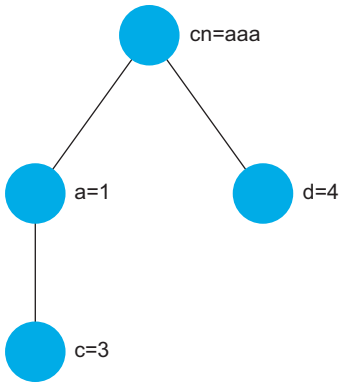

Figure 3 Tree Diagram of Mapping Example

3.1.3 Virtual Object Classes

Virtual object classes are object classes that compose the given virtual entries, determining the virtual attributes that can occur in a virtual entry.

Virtual object classes are defined with the tags `<virtualObjectClass>...</virtualObjectClass>`. These tags must be nested between `<virtualEntry>...</virtualEntry>` and can be repeated. A virtual object class can be structural or auxiliary, just as real object classes. Each virtual entry (either parent or child) must contain one structural virtual object class ([1..1]) and can have as many ([0..N]) auxiliary virtual object classes as needed.

The name of a virtual object class must be defined in the tag attribute `name` (such as `<virtualObjectClass name="oc_1">`). A virtual object class

name must be also defined either in the data view schema or must be an already existing (real) object class in the system.

An example for defining virtual object classes is shown in Example 4.

```
<virtualEntry dn="dc=example,dc=com">
 <virtualObjectClass name="oc_0" />
 <virtualEntry rdn="ou=org">
   <virtualObjectClass name="oc_1">
   </virtualObjectClass>
   <virtualObjectClass name="oc_i">
   </virtualObjectClass>
   <virtualEntry rdn="uid=999342">
     <virtualObjectClass name="oc_k">
     </virtualObjectClass>
     ..
     <virtualObjectClass name="oc_n">
     </virtualObjectClass>
   </virtualEntry>
 </virtualEntry>
</virtualEntry>
```

Example 4 Virtual Object Class Example

3.1.4 Map Rules

Map rules are used to select the source entry and attribute in the core DIT from which the value for the attribute in the virtual entry is retrieved. Map rules are defined with the tags `<mapRule>...</mapRule>`. These tags must be nested between `<virtualObjectClass>...</virtualObjectClass>` and can be repeated as many times ([0..N]) as many real entries are needed to associate attributes from. The core DIT entry name must be defined in the tag attribute `dn`.

The value must be a full core DIT entry DN (such as `<mapRule dn="serv=csps,mscId=<mscId>,ou=multiSCs, dc=example,dc=com">`) or a DN containing variable definition (such as `<mapRule dn="serv=csps, {DE}`). For more information about variable definition, see Section 3.1.6 on page 12.

If attributes from one core DIT entry are used to fill the values of two object classes in a virtual entry, two `mapRule` tags are needed with the same core DIT entry both placed in `virtualObjectClass` tags.

An example for filling two object classes is shown in Example 5.

```
<virtualObjectClass name="oc_1">
  <mapRule dn="serv=EPS,mscId=10,ou=multiSCs,dc=example,dc=com
    ..
  </mapRule>
</virtualObjectClass>
<virtualObjectClass name=" oc_2 ">
    <mapRule dn="serv=EPS,mscId=10,ou=multiSCs,dc=example,dc=com ">
      ..
    </mapRule>
</virtualObjectClass>
```

Example 5 Two Object Classes Filled Example

The associations of real and virtual attributes must be performed within the map rules, with the help of attributes, as described in Section 3.1.5 on page 11.

It is possible to associate all attributes of a core DIT entry to a virtual attribute without defining any attribute rules under a map rule. For that, the tag attribute `allAttributes` must be used with the value `true`. This is useful when all attributes from the core entry are mapped to a virtual object class and have the same names. In this way mapping is simpler which can reduce XML complexity.

Note: The use of the `allAttributes` option causes the following:

- No `<attribute>` tag is allowed within the map rule.
- All of these virtual attribute names must be defined in the current virtual object class and must match attributes in the core DIT.
- Each attribute in this object class in the virtual entry gets its value from the attribute of the same name in the core DIT entry.

An example for defining map rules is shown in Example 6.

```
<virtualObjectClass name="oc_1">
  <mapRule dn="serv=EPS,mscId=10,ou=multiSCs,dc=example,dc=com">
    ..
  </mapRule>
  ..
  <mapRule dn="ei=GPRS,serv=WLAN,mscId=10 ou=multiSCs,dc=example,dc=com" allAttributes="true
  </mapRule>
</virtualObjectClass>
```

Example 6 Map Rule Examples

3.1.5 Attributes

Virtual attributes are attributes of virtual entries that are associated with attributes of real entries. As the result of association, a virtual attribute gets its value from the associated real attribute.

Note: Both multivalued and BLOB (Binary Large Object) attributes are supported in views.

Attribute mappings are placed between the tags `<attribute>...</attribute>`. These tags must be nested between the tags `<mapRule>...</mapRule>` and can be repeated as many times ([0..N]) as many virtual attributes are needed.

The name of the real attribute must be defined in the tag attribute `attrName`, along with its real object class in the tag attribute `objectClass`. Optionally, virtual attributes can be renamed by using the tag attribute `newName`, otherwise the virtual attribute gets its name from the real attribute.

Note: The naming virtual attribute must not be mapped to the attribute in the core entry, because automatically gets value from the virtual entry `dn`.

An example for defining attributes is shown in Example 7.

```
<mapRule dn="serv=EPS,mscId={var_1},ou=multiSCs,dc=example,dc=com">
      <attribute attrName="EpsProfileId" objectClass="EpsStaticInf" />
      <attribute attrName="EpsRoamAllow" objectClass="EpsStaticInf" newName="EpsRoamAllow-v" />
</mapRule>
...
<mapRule
      dn="ei=GPRS,serv=WLAN,mscId={var_1},ou=multiSCs, dc=example,dc=com"
      allAttributes="true">
</mapRule>
```

Example 7 Attribute Examples

3.1.6 Variables

Variables enable the mapping of multiple virtual entries with a single rule. The constant value of a naming attribute can be replaced with a variable (such as `<virtualEntry rdn=" uid=`*`{var_1}`*`">`, where *`{var_1}`* is a variable). In this example, the variable gets its value from the base object in the original LDAP request. Instead of mapping many virtual entries one by one, mapping is done with the usage of variables in a single step.

An example for defining virtual entries with variables is shown in Example 8.

```
<view viewName="AppView">
 <virtualEntry dn="dc=example,dc=com">
   <virtualObjectClass name="oc_0" />
   <virtualEntry rdn="ou=org">
     <virtualObjectClass name="oc_1">
       <virtualEntry rdn="uid={var_1}">
         <virtualObjectClass name="oc_2">
           ..
           <virtualEntry rdn="cn=A">
             <virtualObjectClass name="oc_3">
               <mapRule dn="serv=EPS,mscId={var_1},ou=multiSCs,dc=example,dc=com">
                 ..
               </mapRule>
             </virtualObjectClass>
           </virtualEntry>
         </virtualObjectClass>
       </virtualEntry>
     </virtualObjectClass>
   </virtualEntry>
 </virtualEntry>
</view>
```

Example 8 Virtual Entry Example with Variables

Variable usage assumes that all entries mapped with the same variable have the same structure. In Example 8, all virtual entries that have the `uid` naming attribute are mapped with a single rule, and all of them have the same structure. The '`uid`' value is filled during the reception of the LDAP request with such a naming attribute.

If some entries need different structures from others mapped with a variable naming attribute, define separate virtual entries. These virtual entries have a fixed naming attribute and must be placed above entries with the variable naming attribute, as in Example 9.

```
<view viewName="AppView">
 <virtualEntry dn="ou=org,dc=example,dc=com">
   <virtualObjectClass name="oc_1">
     <virtualEntry rdn="uid=jsmith">
       <virtualObjectClass name="oc_2">
         ..
       </virtualObjectClass>
     </virtualEntry>
     <virtualEntry rdn="uid={var_1}">
       <virtualObjectClass name="oc_3">
         ..
       </virtualObjectClass>
     </virtualEntry>
   </virtualObjectClass>
 </virtualEntry>
</view>
```

Example 9 Virtual Entry Example with Variable and Fixed Value

Mapping stated in Example 9 shows the virtual entry `uid` with fixed (`uid=jsmith`) and variable (`uid=`*`{var_1}`*) naming attribute. If a request with `uid=jsmith` is received, then virtual entry `<virtualEntry rdn="uid=jsmith">` is taken. All other requests containing the `uid` naming attribute take the virtual entry `<virtualEntry rdn="uid=`*`{var_1}`*`">`.

Taking the alias in the core DIT as a source is another way to assign value to a variable. To do so, configure another tag in the `variableDefinition` mapping file. The reason for having such variable usage is that data in the CUDB is stored under internal IDs, but the common way to access subscriber data is by issuing queries that include an external identity. The original DN typically includes an identity and no internal IDs (`mscId` or `assocId`).

To bridge this gap, mapping rules can be extended with a variable receiving value from an alias on the core DIT like this:

```
<virtualEntry rdn="uid={var_1}">
  <variableDefinition variable="DE" aliasedObjectNameInEntry="MSISDN={var_1},dc=msisdn,ou=identities,dc=exampl
</virtualEntry>
```

This mapping rule defines a new variable *`DE`* to store the DN of the entry the alias entry `MSISDN` *`{var_1}`* points to. The variable name is arbitrary (in this example *`DE`*), but it needs to be unique and consistent through the mapping. The value of the variable *`DE`* is stored in the `aliasedObjectName` attribute of the entry specified in `aliasedObjectNameInEntry`.

From that point in the mapping rules onwards, this new variable can be used to build the DNs of the source entries in the core LDAP tree. The variable definition can be used in another variable definition as shown in Example 10.

```
<virtualEntry rdn="uid={var_1}">
  <variableDefinition variable="DE" aliasedObjectNameInEntry="MSISDN={var_1},dc=msisdn,ou=identities,dc=example,d
  <virtualEntry rdn="cn=A">
    <virtualObjectClass name="oc-1">
      <mapRule dn="cn=cnValue-A,serv=CSPS,{DE}">
        ..
      </mapRule>
      <mapRule dn="cn=cnValue-B,serv=EPS,{DE}">
        ..
      </mapRule>
    </virtualObjectClass>
    <virtualEntry rdn="cn=B">
      <variableDefinition variable="CSP" aliasedObjectNameInEntry="ei=GPRS,Serv=CSPS,{DE}"/>
      <virtualObjectClass name="oc2">
        <mapRule dn="{CSP}">
          ..
        </mapRule>
      </virtualObjectClass>
    </virtualEntry>
  </virtualEntry>
</virtualEntry>
```

Example 10 Virtual Entry Example with Variable Definition

In Example 10, the virtual entry `cn=A` is built using the data from core DIT entries, which have the variable *`DE`* in `dn`. To retrieve the core DIT entry, the *`DE`* variable needs to be resolved and filled with the proper value. After that, the core DIT entry has a full DN and can be retrieved from the database. The variable definition has to be defined either in that entry or in one of the ancestors in order to use it. In this case, *`DE`* is defined in the parent `uid`.

The virtual entry `cn=B` is built using data from the core DIT entry with `dn` *`{CSP}`*. This variable is defined in the same entry and it has the value of `aliasedObjectName` in the core entry `ei=GPRS,Serv=CSPS,`*`{DE}`*. To resolve this core entry DN, first the resolving of *`DE`* from one of ancestors is needed. Once *`DE`* is resolved, then DN `ei` is built and used to retrieve the `aliasedObjectName` value that holds the *`CSP`* value in the end.

The core DIT entry `dn` can be defined in the following two ways:

- `<mapRule dn="serv=EPS,mscId=`*`{var_1}`*`,ou=multiSCs, dc=operator,dc=com">`. This approach assumes that the data is accessed through an internal ID (`mscId` or `assocId`), so the core DIT entry `dn` can be built directly.

- `<mapRule dn="serv=EPS, `*`{DE}`*`">`. This approach assumes that data is accessed through an external identity demanding the usage of variable definition that gets value from the core DIT alias (`variableDefinition`).

Accessing entries through alternative names, as in regular aliases, is possible by using `variableDefinition`. Apart from that, `variableDefinition` does not show the aliased DN, but rather keeps DNs in the same branch the request asked for, which is what the aim of alias hiding is.

Accessing data using external identity and internal ID is shown in Example 11.

```
<view viewName="AppView">
 <virtualEntry dn="dc=example,dc=com">
   <virtualObjectClass name="oc_0" />
   <virtualEntry rdn="ou=users">
     <virtualObjectClass name="oc_4">
       ..
     </virtualObjectClass>
     <virtualEntry rdn="userId={mscid_var}">
       <virtualObjectClass name="oc_5" />
       <virtualEntry rdn="cn=B">
         <virtualObjectClass name="oc_6">
           <mapRule dn="serv=EPS,mscId={mscid_var},ou=multiSCs,dc=example,dc=com">
             ..
           </mapRule>
           <mapRule dn="EpsStaInfId=EpsStaInf,mscId={mascid_var},ou=multiSCs,dc=example,dc=com">
             ..
           </mapRule>
         </virtualObjectClass>
       </virtualEntry>
     </virtualEntry>
   </virtualEntry>
   <virtualEntry rdn="ou=people">
     <virtualObjectClass name="oc_1">
       ..
     </virtualObjectClass>
     <virtualEntry rdn="uid={msisdn_var}">
       <variableDefinition variable="DE" aliasedObjectNameInEntry="MSISDN={msisdn_var},dc=msisdn,ou=identities
      dc=example,dc=com" />
       <virtualObjectClass name="oc_2" />
       <virtualEntry rdn="cn=A">
         <virtualObjectClass name="oc_3">
           <mapRule dn="serv=IMS,{DE}">
             ..
           </mapRule>
         </virtualObjectClass>
       </virtualEntry>
     </virtualEntry>
   </virtualEntry>
 </virtualEntry>
</view>
```

Example 11 Mapping Example with External Identity and Internal ID

3.1.6.1 Multiple Variables

Having multiple variables allows the creation of more than one virtual entry using a single definition. Multiple variables are defined in the tags `<virtualEntry>` and `<mapRule>`. Fix values can be replaced by freely chosen variables, must be indicated by {}, and can have any name containing the characters [a..z], [A..Z], [0..9], and [_] (such as `serv={variable_1}`). Variable names must be unique within a full DN.

In case of a child virtual entry, the full DN consists of the parent full DN and the child RDN. To have a valid association between virtual and real entries, the same variable names have to be used both in the associated virtual entry and the map rule.

The variable must be defined at first usage in a virtual entry (such as `<virtualEntry rdn=" vmscId={var_1}">`). The scope and lifetime of a variable is a virtual entry block (opening and closing tags of `<virtualEntry>`), where the variable is defined. The same variable name can be defined again in another virtual entry (at the same level), and can be reused even at a different RDN, but cannot be reused in a child virtual entry. The type of

variables is fetched automatically from the naming attributes they belong to, so a redefinition is not needed.

Note: Regular expressions (such as `uid=223*6`) are not supported as variables.

An example for multiple variables is shown in Example 12.

```
<view viewName="AppView">
 <virtualEntry dn="dc=example,dc=com">
   <virtualObjectClass name="oc_0" />
   <virtualEntry rdn="ou=org">
     <virtualObjectClass name="oc_1">
       <mapRule dn="ou=multiSCs,dc=example,dc=com" allAttributes="true" />
     </virtualObjectClass>
     <virtualEntry rdn="uid={var_3}">
       <virtualObjectClass name="oc_2">
         <mapRule dn="mscId={var_3},ou=multiSCs,dc=example,dc=com">
           <attribute attrName="mscId" objectClass="CUDBMultiServiceConsumer" />
         </mapRule>
       </virtualObjectClass>
       <virtualEntry rdn="v={var_2}">
         <virtualObjectClass name="oc_3">
           <mapRule dn="serv={var_2},mscId={var_3},ou=multiSCs,dc=example,dc=com">
             <attribute attrName="IEG" objectClass="CsPsTRACINGData" newName="IEG-v" />
           </mapRule>
         </virtualObjectClass>
         <virtualEntry rdn="ei-v={var_1}">
           <virtualObjectClass name="oc_4">
           </virtualObjectClass>
           <virtualObjectClass name="alias">
             <mapRule dn="ei={var_1},serv={var_2},mscId={var_3},ou=multiSCs,dc=example,dc=com">
               <attribute attrName="aliasedObjectName" objectClass="alias" newName="ei-virtual-alias" />
             </mapRule>
           </virtualObjectClass>
         </virtualEntry>
       </virtualEntry>
     </virtualEntry>
   </virtualEntry>
 </virtualEntry>
</view>
```

Example 12 Multiple Variables Example

3.1.6.2 Rules for Variables

The rules for the variables are the following:

- The variable names must be unique inside one virtual entry name and inside one core DIT entry name.
- A variable can be used only once inside one virtual entry name and inside one core DIT entry name.
- The variables must be set after each other, from top to bottom in a virtual entry, which equals from right to the left in a core DIT entry. See the following example:

```
<virtualEntry rdn="uid={var_1} ">
  ..
  <virtualEntry rdn="a={var_2}">
    ..
    <mapRule dn="serv={var_2},mscId={var_1},ou=multiSCs,dc=example,dc=com">
      ..
    </mapRule>
    ..
  </virtualEntry>
</virtualEntry>
```

- The position of variables is not relevant. See the following example:

```
<virtualEntry rdn="uid={var_1}">
  ..

  <virtualEntry rdn="a=1">
    ..

    <virtualEntry rdn="b={var_2}">
      ..
      <virtualEntry rdn="c={var_3}">
        ..
        <mapRule dn="m={var_3},e=2,serv={var_2},mscId={var_1},
ou=multiSCs,dc=example,dc=com"><!—notice there is e=2 between var_3 and var_2 -->
          ..
        </mapRule>
        ..
      </virtualEntry>
    </virtualEntry>
  </virtualEntry>
</virtualEntry
```

- If a virtual entry is built using data from several core DIT entries, only variables can be used in all map rules. The user of variables can still be different in the virtual entry, until the rules above apply.

```
<virtualEntry rdn="uid={var_1}">
  ..
  <virtualEntry rdn="a=1">
    ..
    <virtualEntry rdn="b={var_2}">
      <virtualEntry rdn="c=2">
        ..
        <mapRule dn="m=4,e=3,serv={var_2},mscId={var_1},ou=multiSCs,dc=example,dc=com">
          ..
        </mapRule>
        <mapRule dn="k=5,serv={var_2},mscId={var_1},ou=multiSCs,dc=example,dc=com">
          ..
        </mapRule>
        ..
      </virtualEntry>
    </virtualEntry>
  </virtualEntry>
</virtualEntry>
```

3.1.7 Transformation Rules

Transformation rules are used to define a group of bijective (one-to-one) mappings between real (core DIT) and virtual attribute values. These mappings are called attribute value transformations and their usage is described in

Section 3.1.8 on page 19. Each transformation rule has a unique name and an arbitrary number of mappings.

Transformation rules are defined with the *<transformationRule>...</transformationRule>* tags. These tags must be nested between the *<transformationRules>...</transformationRules>* tags and can be repeated as many times ([0..N]) as many transformation rules are needed. The transformation rule name must be defined in the `name` tag attribute. The mappings are nested inside transformation rules, as described in Section 3.1.8 on page 19.

The *<transformationRules>...</transformationRules>* tags must be nested between the *<view>..</view>* tags and there can be only one pair of these tags per view.

An example for defining the transformation rules is shown in Example 13.

```
<view viewName="AppView>
  <virtualEntry dn="cn=aaa,dc=example,dc=com">
    ..
  </virtualEntry>
  <virtualEntry dn="ou=people,dc=example,dc=com">
    ..
  </virtualEntry>
  <transformationRules>
    <transformationRule name="tr_1">
      ..
    </transformationRule>
    <transformationRule name="tr_2">
      ..
    </transformationRule>
  </transformationRules>
</view>
```

Example 13 Transformation Rules Example

One transformation rule can be assigned to one or more attributes whose syntaxes allow the values defined in the rule, but one attribute can be assigned only one transformation rule at a time. Both single-value and multi-value attributes are supported.

An example for assigning a transformation rule to an attribute is shown in Example 14.

```
<mapRule dn="serv=EPS,mscId={var_1},ou=multiSCs,dc=example,dc=com">
   <attribute attrName="EpsProfileId" objectClass="EpsStaticInf" transformationRule="tr_1" />
 </mapRule>
```

Example 14 Assigning Transformation Rule To Attribute Example

Transformation rules allow mapping values between attributes with different syntaxes, in which case different attributes must be defined. For example, if some value is stored as `int` in core DIT, it can be converted to `string`, but only if the virtual attribute has a different name than the core DIT attribute.

An example of using the transformation rule with syntax conversion is shown in Example 15.

```
<mapRule dn="serv=EPS,mscId={var_1},ou=multiSCs,dc=example,dc=com">
 <attribute attrName="AttrInt" objectClass="oc_1" newName="AttrString" transformationRule="tr_int_to_string" /
</mapRule>
```

Example 15 Transformation Rule with Syntax Conversion

3.1.8 Attribute Value Transformation

Attribute value transformation is a one-to-one mapping consisting of a real attribute value and a corresponding virtual attribute value. A group of attribute value transformations forms a single transformation rule, see Section 3.1.7 on page 17 for more details.

Virtual attribute values are only used for virtual data representation and these are never stored in the database. The conversion between real and virtual attribute values is performed on-the-fly during read and write operations. This means that the search operation converts a real value from the database to a virtual value and returns such a value, while the modify operation converts a virtual value from request to a real value and stores such a value to the database. If a transformation for some value is not defined in the rule, then it is used as-is without conversion. It is not possible to define a default mapping for values which are not mapped.

Attribute value transformations are defined with the `<attributeValueTransformation>...</attributeValueTransformation>` tags. These tags must be nested between the `<transformationRule>...</transformationRule>` tags of the transformation rule they belong to and can be repeated as many times ([0..N]) as many transformation value transformations are needed.

The real attribute value must be defined in the `coreDitValue` tag attribute. The virtual attribute value must be defined in the `virtualValue` tag attribute.

An example for defining the attribute value transformations is shown in Example 16.

```
<transformationRules>
  <transformationRule name="intToBoolTrans">
    <attributeValueTransformation coreDitValue="0" virtualValue="false" />
    <attributeValueTransformation coreDitValue="1" virtualValue="true" />
  </transformationRule>
  <transformationRule name="intToColorTrans">
    <attributeValueTransformation coreDitValue="0" virtualValue="red" />
    <attributeValueTransformation coreDitValue="10" virtualValue="white" />
    <attributeValueTransformation coreDitValue="20" virtualValue="blue" />
  </transformationRule>
</transformationRule>
```

Example 16 Attribute Value Transformation Example

3.1.9 One-to-one Mapped Entries

A one-to-one mapped entry is a virtual entry that has exactly the same content as a core DIT entry: the same object classes, the same attributes values, and the same naming attribute. The mapping rule for this kind of entry must include

the onetoone attribute set to yes in the *<virtualEntry>..</virtualEntry>* tags. Also, it must contain the coreDitDn attribute, whose value is the DN of the corresponding entry in the core DIT. A rule, including an object class where the naming attribute is present, needs to be included in one-to-one mapped entries for consistency reasons. See Example 17.

```
<virtualEntry rdn="serv=AAA" onetoone="yes" coreDitDn="serv=AAA,{DE}">
 <virtualObjectClass name="CUDBService">
 </virtualObjectClass>
</virtualEntry>
```

Example 17 One-to-one Mapped Entries

Note: *DE* is a variable definition in a view. For more information, see Section 3.1.6 on page 12.

One-to-one mapped entries and the corresponding core DIT entries must have the same naming attribute, as it can be seen in Example 17 (serv=AAA).

3.1.10 Scaffolding Virtual Entries

Scaffolding virtual entries are virtual LDAP entries with no content except the naming attribute. They exist for the structure of the virtual LDAP tree and for support only: these entries are neither PLDB nor DSG entries and the views engine does not need to access any database cluster to build them. See Example 18.

```
<virtualEntry rdn="cn=empty" >
 <virtualObjectClass name="oc-1">   <!-- oc-1 includes the 'ou' attribute -->
 </virtualObjectClass>
</virtualEntry>
```

Example 18 Scaffolding Virtual Entry

The scaffolding virtual entry must have one structural object class with only one MUST attribute that holds the virtual naming attribute. The scaffolding virtual entry can also hold variableDefinition if needed for subentries to fill their variables, as seen in Example 19.

```
<virtualEntry rdn="uid={var}">
  <virtualObjectClass name="oc-2">   <!-- oc-2 includes the 'uid' attribute -->
  </virtualObjectClass>
  <variableDefinition variable="DE" aliasedObjectNameInEntry="MSISDN={var},dc=msisdn,ou=identities,dc=example,dc=
</virtualEntry>
```

Example 19 Scaffolding Virtual Entry with Variable Definition

3.1.11 Defining an Alias

An alias is a special attribute with special mapping. A virtual alias entry must include the alias object class (for example, there must be a <virtualObjectClass name="alias"> directive) and a mapping rule is also needed to provide a value for the aliasedObjectName virtual attribute. An example is shown in Example 20.

```
<virtualEntry rdn="uid={var_1}">
  <virtualObjectClass name="oc_1">
    <mapRule dn="mscId={var_1},ou=multiSCs,dc=example,dc=com">
      ...
    </mapRule>
  </virtualObjectClass>
  <virtualEntry rdn="cn=aliasEntry">
    <virtualObjectClass name="oc_2">
    </virtualObjectClass>
    <virtualObjectClass name="alias">
      <mapRule dn="serv=csps,mscId={var_1},ou=multiSCs,dc=example,dc=com">
        <attribute attrName="aliasedObjectName" objectClass="alias" />
      </mapRule>
    </virtualObjectClass>
  </virtualEntry>
</virtualEntry>
```

Example 20 Alias Example

Just like regular (non-views) alias entries, virtual alias entries need at least one supporting object class to hold the naming attribute. A virtual alias can get a new name as well as other virtual attributes.

Virtual aliases point to virtual entries, whose DN is defined in the `aliasedObjectName` attribute. The `aliasedObjectName` virtual attribute must be filled with a value of a valid virtual DN, which must be stored somewhere in the core DIT.

A DSG-to-DSG alias must point to a virtual entry under the same Distribution Entry (DE). Otherwise, an error 36 is returned when trying to dereference such an alias. Also, a DSG-to-DSG alias with a variable definition must not point to an entry that does not have the same variable definition and conversely.

Refer to *CUDB LDAP Data Views*, Reference [2] for more information.

3.2 Configuration

To configure an LDAP data view, perform the following steps:

1. Prepare and upload the configuration files. The data view schema and the mapping file have to be stored in the `/home/cudb/dataAccess/ldapAccess/ldapFe/config/views` directory.

   The schema for one-to-one entries is handled like any other application schema. For detailed information, refer to *CUDB Node Schema Update*, Reference [3].

2. Validate configuration files with the Checking Tool. For more information, see Section 3.3 on page 24.

3. Create the LDAP data view, see Section 3.2.1 on page 22. On every CUDB node, perform the procedure described in Section 3.2.1 on page 22.

4. Assign the LDAP data view to the user, see Section 3.2.2 on page 22.

To delete an LDAP data view, perform the following steps:

1. Unassign the LDAP data view from user, see Section 3.2.3 on page 23.

2. Delete the LDAP data view, see Section 3.2.4 on page 23.

When the data view schema or the mapping files are modified, the LDAP FEs must be restarted manually by executing the `cudbLdapFeRestart` command.

After the new configuration is activated, check if the LDAP FEs are up and running. For possible errors and solutions, refer to *CUDB Troubleshooting Guide*, Reference [5].

Note: Configuration changes related to these procedures will be rejected if the Application Facilitator Value Package is not available.

3.2.1 Creating LDAP Data View in the Data Model

To create an LDAP data view, add an instance of `CudbLdapView` class in the configuration model. For more information, refer to the *Class CudbLdapView* section in *CUDB Node Configuration Data Model Description*, Reference [4].

Refer to the Object Model Modification Procedure in *CUDB Node Configuration Data Model Description*, Reference [4] for more information on all the steps required to modify the object model (for example, on using the `applyConfig` administrative operation to activate the changes).

3.2.2 Assigning LDAP Data View to User

To assign an LDAP data view to the user, set the `cudbLdapViewId` attribute of the already created instance of the `CudbLdapUser` class. For more information, refer to the *Class CudbLdapUser* section in *CUDB Node Configuration Data Model Description*, Reference [4].

The value of the `cudbLdapViewId` attribute is the identifier of the LDAP data view connected to the user, and its value must match the corresponding value of the `cudbLdapView` class. For more information, refer to the *Class CudbLdapView* section in *CUDB Node Configuration Data Model Description*, Reference [4].

Refer to the Object Model Modification Procedure in *CUDB Node Configuration Data Model Description*, Reference [4] for more information on all the steps required to modify the object model (for example, on using the `applyConfig` administrative operation to activate the changes).

Update the information of the LDAP users on every CUDB node apart from the one selected in this section by following the procedure described in the *Updating CUDB LDAP User Information in a CUDB Node* section in *CUDB System Administrator Guide*, Reference [1].

Note: The same LDAP data view can be assigned to several LDAP users. An LDAP user can either use a view or see the core DIT. An LDAP user can have only one view assigned.

Although, it is not allowed to assign an LDAP view to a provisioning (`isProvisioningUser=TRUE`) or to a re-provisioning (`isReProvisioningUser=TRUE`) user.

3.2.3 Removing LDAP Data View from User

To remove the LDAP data view from the user, remove the value of the `cudbLdapViewId` attribute of the instance of the `CudbLdapUser` class. For more information, refer to the *Class CudbLdapUser* section in *CUDB Node Configuration Data Model Description*, Reference [4].

Refer to the Object Model Modification Procedure in *CUDB Node Configuration Data Model Description*, Reference [4] for more information on all the steps required to modify the object model (for example, on using the `applyConfig` administrative operation to activate the changes).

Update the information of the LDAP users on every CUDB node apart from the one selected in this section by following the procedure described in the *Updating CUDB LDAP User Information in a CUDB Node* section in *CUDB System Administrator Guide*, Reference [1].

Note: If an LDAP user has no view assigned, it sees the core DIT.

3.2.4 Deleting LDAP Data View

To delete an LDAP data view, remove the instance of the `CudbLdapView` class from the data model. For more information, refer to the *Class CudbLdapView* section in *CUDB Node Configuration Data Model Description*, Reference [4].

Refer to the Object Model Modification Procedure in *CUDB Node Configuration Data Model Description*, Reference [4] for more information on all the steps required to modify the object model (for example, on using the `applyConfig` administrative operation to activate the changes).

Note: Only LDAP data views that are not connected to any user can be deleted.

After an LDAP data view is deleted through the data model, also delete the view-specific mapping file and schema files stored in the `/home/cudb/dataAccess/ldapAccess/ldapFe/config/views/` directory.

Note: If an LDAP data view contains one-to-one mapped entries, the view cannot be deleted completely. All written changes (added one-to-one mapped entries, modified entries) remain in the core DIT.

3.2.5 Modifying LDAP Data View

An existing LDAP data view can also be modified. The mapping configuration file, the data view schema or permissions can be changed. These changes imply running the Checking Tool to ensure valid configuration files. For more information, see Section 3.3 on page 24.

Adding, deleting or changing attributes and object classes, changing the tree and changing the mapping can easily be performed. After any of these changes, if they are valid and well-formed (as described in earlier chapters), just restart the LDAP FEs and all changes are applied and ready to use.

New view-related attributes and object classes can also be added to the core DIT: create the new schema files or update the existing ones as described in *CUDB Node Schema Update*, Reference [3]. After the schema update, new attributes and object classes can be used through the view.

3.3 Checking Tool

The sections below contain the instructions to validate CUDB LDAP Data Views mapping and schema files against a given set of rules. After the view mapping and schema files are prepared, validation is necessary before views configuration. The tool executes semantic and syntactic checks for the mapping and schema files of the different LDAP data views. It also performs the validation of mapping files, and checks if they are defined according to a set of rules assuring that data view mappings are correctly configured. Use the tool before view configuration in the data model, and after view mapping file modifications. However, the Checking Tool ensures that the view mapping file is properly written, the static checks of the tool do not check the existence of entries.

Note: It is mandatory to run the Checking Tool in order to validate mapping and schema files, and therefore avoid LDAP FE runtime errors.

3.3.1 Input

The input files for the Checking Tool are XML mapping files and schema files. The tool searches and performs checks in the folder containing these files. An XSD file (`mapping.xsd`) is provided with the Checking Tool, serving as an input file to validate the XML structure of the mapping.

3.3.2 Execution

To start the Checking Tool, enter the installation directory of the tool on the node and execute the following script:

```
cd /opt/ericsson/cudb/ldapviews/bin

./cudbCheckLdapViewMapping.sh
```

The output must be similar to the following:

```
#####################################################
#                                                   #
#                  WELCOME!                         #
#         Consistency checker tool for              #
#         LDAP Schema & XML mapping files           #
#                                                   #
#####################################################
Searching directory /home/cudb/dataAccess/ldapAccess/ldapFe/config/views for .xml and .schema files...
Searching directory /opt/ericsson/cudb/ldapviews/bin. for .xsd file...
Searching directory /opt/ericsson/cudb/ldapviews/bin. for .conf file...

Found pairs: {Test2.xml=Test2.schema}
Found config file: config.conf
Found XMLschema file: mapping.xsd

PROCESSING :
./Test2.schema and ./Test2.xml
```

3.4 Output

The following outputs are obtained as the result of tool execution:

- If an error is detected at command execution, the execution continues until all rules are checked, and an error message is displayed for each case:

  ```
  CHECKING RULES:
  RULE: "dn" attribute in VirtualEntry xml tag must be
  present only in topmost VirtualEntry, inner Virtual
  entries must have only "rdn" attributes...  OK.
  RULE: if "map all attributes" is present, no further
  attribute mappings for that maprule are allowed...  OK.
  RULE: if "new name" attribute is defined in mapping
  file, it may not be empty...  OK.
  RULE: rdn attribute in mapping file can not be
  multi-value...  OK.
  RULE: for "new name" attrs check that the value we
  provide fit the syntax of the attribute (including
  attribute overloading) ...  OK.
  RULE: syntax(multiplicity) of virtual attribute matches
  syntax of mapped attribute...  OK.
  RULE: if naming attribute is defined as variable, it
  must be used somewhere in the DN of the source
  ```

- If there is a syntax error in the file, a message indication of the error found is displayed. An example of such an error found is the following:

  ```
  Validating mapping file TestView.xml...
  Validation error:  org.xml.sax.SAXParseException:
  Element type "virtualObjectClass" must be followed by
  either attribute specifications, ">" or "/>".
  ```

After running the tool, one of the following results can be obtained:

- If the command completion is successful and no errors are found by the tool, the following message is displayed:

  ```
  Validating mapping file TestView.xml...OK.
  Processing mapping file TestView.xml...OK.
  Processing schema file TestView.schema...OK.
  ```

- If the tool detects some inconsistencies and broken rules, an error message is displayed as described in the beginning of this section.

  For example, if virtual attributes or virtual object classes are defined in the mapping but not defined in the schema, or are defined only in the schema but not defined in the mapping, it generates warnings and the following messages are displayed:

  ```
  Validating mapping file TestView.xml...OK.
  Processing mapping file TestView.xml...OK.
  Attributes of object class ocTest1 found in schema file
  TestView.schema but not in mapping file TestView.xml:
  testAttr1
  testAttr2
  ```

4 Appendix: Mapping Structure

See Example 21 for a formal definition of the general structure of a mapping file.

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
<xsd:element name='view'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='virtualEntry' maxOccurs='unbounded'/>
    <xsd:element ref='transformationRules' minOccurs='0' maxOccurs='1'/>
   </xsd:sequence>
   <xsd:attribute name='viewName' type='xsd:string' use='required'/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='virtualEntry'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='virtualObjectClass' maxOccurs='unbounded'/>
    <xsd:element ref='virtualEntry' minOccurs='0' maxOccurs='unbounded'/>
 <xsd:element ref='variableDefinition' minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
   <xsd:attribute name='dn' type='xsd:string' use='optional'/>
   <xsd:attribute name='rdn' type='xsd:string' use='optional'/>
   <xsd:attribute name='onetoone' type='xsd:string' use='optional'/>
   <xsd:attribute name='coreDitDn' type='xsd:string' use='optional'/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='variableDefinition'>
  <xsd:complexType>
  <xsd:simpleContent>
    <xsd:extension base='xsd:string'>
     <xsd:attribute name='aliasedObjectNameInEntry' type='xsd:string' use='required'/>
     <xsd:attribute name='variable' type='xsd:string' use='required'/>
    </xsd:extension>
   </xsd:simpleContent>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='virtualObjectClass'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='mapRule' minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
   <xsd:attribute name='name' type='xsd:string' use='required'/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='mapRule'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='attribute' minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
   <xsd:attribute name='dn' type='xsd:string' use='required'/>
   <xsd:attribute name='allAttributes' type='xsd:string' use='optional'/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='attribute'>
  <xsd:complexType>
   <xsd:simpleContent>
    <xsd:extension base='xsd:string'>
     <xsd:attribute name='attrName' type='xsd:string' use='required'/>
     <xsd:attribute name='objectClass' type='xsd:string' use='required'/>
     <xsd:attribute name='newName' type='xsd:string' use='optional'/>
     <xsd:attribute name='transformationRule' type='xsd:string' use='optional'/>
    </xsd:extension>
   </xsd:simpleContent>
  </xsd:complexType>
```

```
 </xsd:element>

 <xsd:element name='transformationRules'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='transformationRule' minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='transformationRule'>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref='attributeValueTransformation' minOccurs='0' maxOccurs='unbounded'/>
   </xsd:sequence>
   <xsd:attribute name='name' type='xsd:string' use='required'/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name='attributeValueTransformation'>
  <xsd:complexType>
   <xsd:attribute name='coreDitValue' type='xsd:string' use='required'/>
   <xsd:attribute name='virtualValue' type='xsd:string' use='required'/>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Example 21 Formal Definition of the General Structure of a Mapping File

Glossary

For the terms, definitions, acronyms and abbreviations used in this document, refer to *CUDB Glossary of Terms and Acronyms*, Reference [6].

Reference List

CUDB Documents

[1] *CUDB System Administrator Guide*

[2] *CUDB LDAP Data Views*

[3] *CUDB Node Schema Update*

[4] *CUDB Node Configuration Data Model Description*

[5] *CUDB Troubleshooting Guide*

[6] *CUDB Glossary of Terms and Acronyms*