Specification DSP0100

Guidelines for CIM-to-LDAP Directory Mappings

May 8th, 2000

Abstract

This paper discusses issues involved with mapping CIM schema to directories that support the LDAP protocol and the X.500 model, and provides guidelines for resolving the issues. Several directory schema entities that are useful in implementing mappings are defined. This is a living document that embodies the collective experience of the DMTF-LDAP working group.

Change History

Version	Date	Description
0.7	4/26/2000	Final edits for submission for member review
0.6	4/4//2000	Incorporated comments form user/security working group, including adding a section on use of existing LDAP classes
0.5	3/31/2000	Incorporate more working group comments
0.4	3/28/2000	Incorporate working group comments
0.3	3/6/2000	Initial review

Author

Doug Wood, Tivoli Systems Inc.

Acknowledgment

This work is a product of the DMTF LDAP Mapping Working Group and has benefited from many comments and discussions during this groups meetings.

Table of Contents

1 Introduction

The Common Information Model (CIM) provides a mechanism for modeling various types of information. The model is independent of any implementation or repository. For a model or *schema* to be useful, it must be mapped into some implementation. This paper discusses issues involved with mapping, and provides guidelines to aid in mapping CIM schema to directories that support the LDAP protocol and the X.500 model.

There are several incongruities between the CIM meta model and the X.500 model. For a successful mapping to be implemented, these incongruities must be resolved. The most significant differences are:

- Object identification,
- Associations, and
- Data types.

Object Identification

There are differences in both the composition and scope of uniqueness of the name domains between CIM and the X.500 model. The CIM name space must be projected onto the directory name space in a way that preserves unique identification of every CIM instance.

Associations

The Common Information Model supports a formal relationship model in which all relationships between classes are expressed as association classes. The X.500 model provides a hierarchical structuring of data elements, but no other means of expressing relationships is explicitly supported. One or more mechanisms must be provided to map CIM associations to directories.

Data Types

Not all data types defined by CIM can be directly mapped to syntaxes supported by the LDAP protocol.

2 Object Identification

Instances of CIM classes are uniquely identified by their key properties, their class name, and a namespace identifier. The set of key properties must be unique for all instances of the class and any sub classes. Directory entries are uniquely identified by their distinguished name, which also defines their location in the directory tree. Each component of the distinguished name must be unique for all entries of any type that share the same parent. Directories use one or more attributes of each entry to form its Relative Distinguished Named or RDN.

There must be an algorithmic means of mapping the identity of any CIM object uniquely into a directory, and of deriving the CIM identification of any previously mapped directory entry.

2.1 CIM Namespace

The namespace identifies the type and instance of a system hosting a CIM implementation. All CIM entities hosted by the same implementation share the same namespace. This paper assumes that a single directory instance hosts no more than one CIM implementation. Therefore, the namespace is the same for all CIM instances stored in the directory and is not useful for identification within the context of the mapping.

Future versions of this paper may address a directory instance hosting more than one CIM implementation.

2.2 RDN Attributes

There are several issues that arise when selecting the RDN for mapped entries. They include:

1. CIM classes may have multi-valued keys, but many directory servers have problems with multi-attribute RDNs.
2. The scoping rules for CIM keys and directory RDNs are different. CIM keys must be unique for all instances of the class in which a key property is declared. RDNs must be unique for all entries with a common parent. It is possible for instances of different classes to have the same key value, to be stored under the same parent.

To resolve both of these issues and to insure consistent naming of mapped CIM instances, the following mechanism for creating RDNs is recommended.

The value of the RDN is one of two variants of the CIM Model Path. The CIM Specification defines a *Model Path as*

> *The object name constructed as a scoping path through the CIM schema is referred to as a Model Path. A model path is a combination of the key properties values qualified by the class name. It is solely described by CIM elements and is absolutely implementation-independent. It is used to describe the path to a particular object or to identify a particular object within a namespace. The name of any object is a concatenation of named key property values, including all key values of its scoping objects. When the class is weak with respect to another class, the model path includes all key properties from the scoping objects*[1].

The model path is defined to be unique across all instances of all CIM classes within a namespace; therefore, it provides the basis for assigning RDN values. The model path definition is not complete for our purposes in that it does not specify the order of values for classes with multi-valued keys. To resolve the ambiguity, and to avoid problems with multi-attribute RDNs, two new attributes, orderedCimKeys and orderedCimModelPath, are defined for mapping purposes. They are defined as:

orderedCimKeys

```
(     1.3.6.1.4.1.412.100.1.2.1
      NAME        'orderedCimKeys'
      DESC        'The model path for the instance (without
                  propagated keys). May be used as an RDN'
      SYNTAX      DirectoryString
      SINGLE-VALUE
      EQUALITY    octetStringMatch
)
```

orderedCimModelPath

```
(
      1.3.6.1.4.1.412.100.1.2.2
      NAME        'orderedCimModelPath'
      DESC        'The model path for the instance (with propagated
                  keys). May be used as an RDN'
      SYNTAX       DirectoryString
      SINGLE-VALUE
      EQUALITY    octetStringMatch
)
```

The value of these attributes are constructed by ordering the CIM keys [formatted as "**<className>**.**<key>=<value>**[,**<key>=<value>**]*"] of the object in the US-ASCII collation order of the property names. Propagated keys are not included in orderedCimKeys, but are included in orderedCimModelPath. Consider, as an example, the case of an instance of the class CIM_Rack. It's Model Path identification might be:

```
CIM_Rack.CreationClassName=CIM_Rack,Tag=AcmeRack4279B
```

where the ordering of the keys is not significant. The corresponding value of orderedCimModelPath and orderedCimKeys, which happen to be the same for CIM top-level classes, is:

```
CIM_Rack.CreationClassName=CIM_Rack,Tag=AcmeRack4279B
```

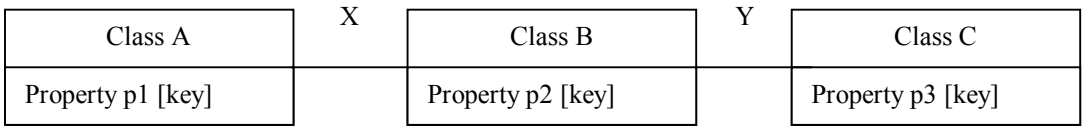
where the ordering of the keys *is* significant.

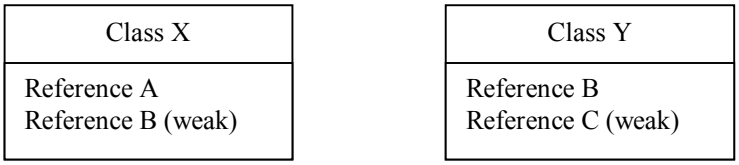
The attribute orderedCimKeys is used as the RDN for all CIM classes mapped to a directory, with the possible exception of the weak side of weak associations, where, as described below, in certain circumstances, orderedCimModelPath is used.

2.3 RDN Attribute for Weak Associations

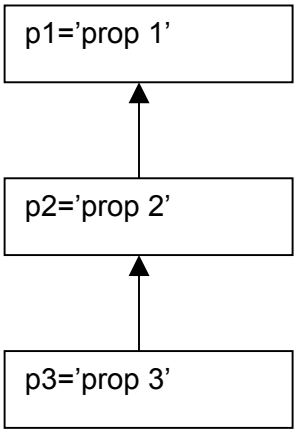
When weak associations are mapped through DIT containment, the RDN can benefit from special treatment to reduce its complexity. Because the key propagation method of weak associations is similar to the way a DN is assembled, the RDN of the weak member of the association only needs to contain the key properties that it adds.

An example illustrates this. Consider the following set of CIM classes:





Now consider instances of these classes with the associations mapped by DIT containment.



Using orderedCimModelPath as defined above, which includes propagated keys, the RDN for the three entries would be:

```
orderedCimModelPath=A.p1=prop 1
orderedCimModelPath=B.p1=prop 1, p2=prop 2
orderedCimModelPath=C.p1=prop 1, p2=prop 2, p3=prop 3
```

The similarity between the model path for weak associations and a DN is obvious. Therefore, for entries on the weak side of a weak association, the RDN needs only to contain the properties added to the key by the entry, and can be expressed as a value of orderedCimKeys. The remaining properties are available in the DN. Using this approach, the three RDNs are:

```
orderedCimKeys=A.p1=prop 1
orderedCimKeys=B.p2=prop 2
orderedCimKeys=C.p3=prop 3
```

Moreover, abbreviating for the attribute name, the DN for the entry of class C is:

```
ock=C.p3=prop 3, ock=B.p2=prop 2, ock=A.p1=prop 1
```

This is the usage defined for orderedCimKeys; that is, propagated keys are not included in its value. If this behavior is not desired, the alternate naming attribute

orderedCimModelPath may be used. It is defined exactly the same as orderedCimKeys except that all propagated keys must be present.

If weak associations are not mapped by DIT containment, then the propagated keys are not easily inferred from the directory structure, and may be required to insure the uniqueness of RDNs. In this case, orderedCimModelPath should be used for the RDN. For classes that do not participate in weak associations, and therefore do not have propagated keys, orderedCimKeys and orderedCimModelPath are equivalent. Therefore, if orderedCimModelPath is required by weak associations, it may be used for all RDNs. Note, however, that not using DIT containment to map weak associations may cause interoperability issues.

3 Association Mapping

The Common Information Model uses *association* classes to express relationships between instances of classes in the model. The design of associations resembles the OMG CORBA relationship service. That is, an association allows two previously unrelated classes to be associated without modifying either of them. This is done by creating a new association class, which has references to each of the classes to be associated. Association classes are the only classes that may contain references. A reference is simply a pointer to an object instance. Its value is the key (model path) of the instance to which it points. References may not be arrays. This implies that there is an instance of an association for every related object pair (or set if the cardinality of the association is greater than two). For example if instance A is related to instances X, Y, and Z by the same association, there must be three instances of the association AX, AY, and AZ.

When CIM associations are mapped to a backing store, it is desirable that they exhibit the following characteristics:

- They should be stable.
- They should allow efficient traversal, using mechanisms native to the backing store.
- They should not adversely affect the scalability or performance of the backing store.
- The mapping should not alter the semantics of the information model, or introduce additional semantics.

Stability

Stability implies that the values of association references are always valid. This is always an issue with object deletes. Ideally, any associations referencing the deleted object are also deleted. Depending on how the associations are stored, it may be acceptable to simply null out the pointer. Where it is implied by the semantics of the association, it is desirable for the backing store to perform cascade deletes.

For some backing stores, such as directories that have structured storage, move or rename operations can also cause problems. When an association pointer uses the storage location (DN) to reference the associated object, then it must be updated if the object is renamed or moved. Many directories do not guarantee referential integrity of DN syntax attributes by making the update automatically.

Efficient Traversal

Because most implementations do not bring the entire information model into memory, associations are generally traversed in the context of the backing store, using mechanisms provided by it. In the case of directories where search efficiency is tied to directory structure, the mechanism used to store associations has a large impact on traversal efficiency.

Scalability

The mechanism used to store associations can affect scalability and performance in several ways. These include:

- Number of entries

  Because CIM models each association instance as a unique object instance, associations have the potential to increase by several times the total number of entries that must be stored. For an entry-centric store such as a directory, this effectively reduces the size model that can be stored by the same factor.

- Search performance

  The mechanism used to store the object reference in the association should play into the strengths of the backing store search engine. In the case of directories, search efficiency is dependent upon directory structure and object location.

- Number of lookups

  If following an association requires several individual searches, even if each individual request is very efficient as viewed from the backing store, the network overhead of making each request could result in poor client performance.

Semantics

Semantics is not much of a problem for a flat data store such as an RDBMS, but for a structured data store such as a directory, it must be considered. In a directory, the location an entry is stored at has meaning. It implies a relationship with entries that are stored above and below it. Ideally, the implied relationship will reinforce the semantics of the information model, not contradict it.

3.1 Mapping Strategies

There are three general approaches that can be utilized to map CIM associations to a directory. They are:

- Storing an entry reference,
- Using directory tree structure to express associations, and
- De-normalizing the information model.

Within that framework, there are a number of variations and hybrids. Each of the three main approaches has advantages and disadvantages, which vary based on:

- The semantics of the association being mapped,
- The target directory, and

- The performance characteristics of the target application.

Each approach is evaluated against the criteria discussed above.

3.1.1 Entry References

The pointer in the association class can be stored as a reference to the entry that is referenced by the association. There are two ways that the reference may be expressed:

- As the DN of the entry, or
- As the value of a globally unique required value of the entry.

There are several ways that the references may be used. All uses share common characteristics of the reference mechanism.

DN Pointers

Advantages

- It can model any association, and introduces no additional semantics.
- For many servers, retrieval of an entry by its DN is a very vast operation.
- It is simple from both a conceptual and implementation view.

Disadvantages

- Because many directories do not provide support for maintaining the integrity of DN pointers, they are not stable. Deletion of entries leaves dangling references. Moving or renaming entries creates broken references.

Unique Key Reference

In this approach, the reference properties are attributes that store the value of some unique property of the referenced class. There are three possibilities for the unique property.

- The CIM key,
- The directory RDN. and
- An additional unique key value provided for all directory entries.

There are problems with guaranteeing the universal uniqueness of either the CIM key or the RDN, so these values may not be suitable. (See the discussion of RDN construction in section 2.2) There are several algorithms for generating unique key values or Universally Unique IDs (UUIDs). It would be straightforward to add a Must attribute to all directory entries which map to CIM classes. This might be done by adding it to the mapping of ManagedElement that all CIM classes are derived from. Some directories include a UUID in their definition of top or as an operational attribute. This approach may be particularly appealing for these directories.

UUIDs provide stability for move and rename operations because a search for the UUID value will always return the correct entry no mater where it is moved to, or what it is named.

If orderedCimModelPath is used for the RDN, it may be used as the unique identifier because CIM guarantees that the model path is unique for any object instance.

Advantages

- It can model any association, and introduces no additional semantics.
- It may be more stable than DN pointers because moving and renaming entries is. a stable operation

Disadvantages

- Deletes can result in dangling entries.
- Retrieval may be slow.
- Some may find the unique key an intrusive requirement.

3.1.2 DIT Containment

This approach recognizes that the parent-child relationship of the directory tree is a relationship explicitly managed by directories. Associations are expressed by placing the related classes in a parent-child relationship in the directory tree.

Advantages

- Because the relationship is explicitly supported by directories, it is completely stable. It supports cascade deletes.
- Traversal of the association is fast and easy.

Disadvantages

- Many-to-many relationships can not be modeled.
- Only one association per instance may be expressed through the parent-child relationship.
- The semantics of containment expressed by the parent-child relationship is not appropriate for all associations.
- The semantics of the directory tree are already overloaded by directory servers. It is used for administration of access control, and for partitioning. Forcing the structure to express associations could interfere with its other uses.

  For example, There might be an association between a computer system and a location expressing the physical location of the system. If that association is mapped by DIT containment, then systems are stored beneath the location at which they reside. If a particular implementation restricts access to systems based on owning organization, then the most efficient DIT structure for administrative purposes is to place systems beneath organizations. However, this structure is prohibited by the requirements of the association mapping.

3.1.3 De-normalization

De-normalization involves grouping data elements that are normally independent into a single instance. In cases where a data element is referenced by more than one entity, the data element may need to be duplicated when the references are de-normalized.

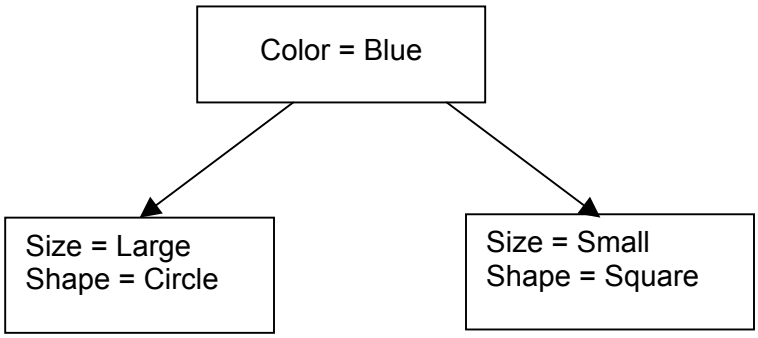
General approaches

The CIM schemas are highly normalized, whereas directory schemas are typically not at all normalized. To make a CIM schema more compatible with a directory, it can be de-normalized. There are two possible levels of de-normalization:

- Association classes can be eliminated
- The association class and all but one of the associated classes can be folded into the remaining class

To implement the de-normalization, the attributes of the associated classes must be combined to form a new de-normalized class. This works well for one-to-one relationships. Several specific mechanisms are examined following this discussion. One-to-many relationships are more difficult because directories provide no formal means of defining structured or aggregate attributes, and multi-valued attributes are not ordered. It is not practical to de-normalize many-to-many associations.

To illustrate one-to-many, consider classes A and B, which have a one-to-many association from A to B defined as C. Class A has the single attribute Color. Class B has the attributes Size and Shape. For simplicity, class C has no attributes other then the references.



When association C is de-normalized, the resulting class D has the attributes Color, Size, and Shape. The instance resulting from the above example is

```
Color = Blue
Size  = Large
      = Small
Shape = Circle
      = Square
```

Because multi-valued attributes are not ordered it could just as easily be:

```
Color = Blue
Size  = Small
      = Large
Shape = Circle
      = Square
```

It is not possible to determine the original grouping of attribute values.

Combining attributes

For a one-to-one association, the attributes of the two classes may be combined into a new class. There are three approaches that may be used to accomplish this. They are:

- A single new class may be declared that has the attributes of both classes.
- The two classes may be layered into a single class using inherence. This provides documentation of the original source of each attribute, and it places the name of both classes into the value of the objectClass attribute. (See section 5.4.)
- An auxiliary class may be declared with the attributes of one of the associated classes and applied to entries that participate in the association.

Advantages

- Its stable – Both move safe and delete safe.
- Retrieval is fast.

Disadvantages

- Not all directory servers allow the objectClass attribute to be modified. On these servers, associations could only be added when the entry is created, or in one case, when the auxiliary class is defined.
- Because an association simply adds attributes to an entry, only a single instance of an association type can be expressed through an auxiliary class. (one-to-one relationship)
- The set of attributes in the classes to be associated must be disjoint. This is only guaranteed if directory attributes are never used to map more than one CIM class.
- The semantics of the auxiliary class imply that one member of the association is dominant. Not all associations have this semantic.
- It can not be used to associate two instances of the same class.

Formatted string

A solution to de-normalizing one-to-many associations used by several commercial directory applications is to store all of the attributes of the subordinate class in a single formatted string. Using this approach Class D, from the previous example, might have attributes Color, and ShapeDescription. The instance would be stored as

```
Color             = Blue
ShapeDescription  = Large - Circle
                  = Small - Square
```

Advantages

- Search and retrieval are fast.
- Traversal of the association is fast and easy.
- It is stable.

Disadvantages

- Formatted strings require client applications to understand the structure of the data.
- Formatted strings add complexity to the client because of the requirement to parse and build the strings.
- De-normalization implies a superior–subordinate relationship that may not be appropriate for all associations.
- If a class participates in more than one association that is de-normalized, then data must be duplicated.

3.1.4 Summary

As evidenced by the listed disadvantages, there is no single general-purpose solution. Both DIT containment and de-normalization may work well for relationships with strong containment semantics, but may not be suitable for more loosely associated classes. In addition, a class may only participate as a subordinate in one association that is mapped through either of these mechanisms. Entry references are the only general-purpose mechanism available, but they are not well supported on many directory servers. They may be used in combination with a limited use of de-normalization to eliminate the association classes.

3.2 Weak Associations

Despite the disadvantages of directory containment, there is a class of associations well suited to mapping in this way because its semantics are similar to the directory containment relationship. Weak associations express the same parent-child relationship as the directory tree structure, and the life cycle of the child is linked to the life cycle of the parent. The types of entities modeled by weak associations are not likely to conflict with either the administrative or the partitioning function of the directory tree. In both cases, the parent and related classes can be treated as a single entity.

If the weak association has no attributes, it does not need to be stored. The parent-child relationship of the entries expresses everything required. If the association has attributes, the attributes must be stored. There are two possibilities.

- The association could be stored as a separate entry as a child of the dominant class. Other referenced classes are stored as its children. This adds to complexity and retrieval time by adding an addition layer in the directory tree. However, it is the only mechanism that supports weak associations of greater cardinality then binary.
- The properties of the association can be stored with the child entry by use of an auxiliary class. Because an instance can only be weak with respect to one other instance, there will never be a need to store more than one instance of the association class with the child.

3.3 Use of Auxiliary Classes

Auxiliary classes provide a mechanism for adding additional attributes to an entry without modifying its structural class. As such, their use in mapping associations represents some variant of de-normalization. Because of their flexibility, a number of the mechanisms listed utilize them.

There is some variance among directory servers in their implementation of auxiliary classes. For simplicity, this document assumes they are implemented in accordance with the X.500 specification. In particular, auxiliary classes are added to entries on a per-entry basis, and existing entries may be modified to include an auxiliary class. It is left to the implementor to modify suggested mechanisms to conform to the actual implementation of a target directory server.

The use of auxiliary classes to add associations to entries preserves the intent of the association mechanism. Auxiliary classes allow new associations to be defined for existing entries without requiring their definition to be modified.

3.4 Catalog of Mechanisms

Following is an (incomplete) list of possible mechanisms using combinations of the approaches discussed previously.

1. Store the association as a separate class exactly as it is represented in the CIM schema. Use references for the reference properties.
2. Use an auxiliary class to add a reference to one of the classes referenced by the association. The association class is de-normalized out of existence.
3. Use an auxiliary class with two references that is added to both members of the association. This allows the association to be easily traversed from either direction. The association class is de-normalized out of existence.
4. Add the reference attributes to one or both classes in the association. The Association class is de-normalized out of existence. If the references attribute is not used by an instance then the association doesn't exist.
5. Create a class that has the references required by the association, and a class name unique to the association. An instance of the class is stored by DIT containment under each participant in the association. The references point to all members of the association.
6. Use DIT containment to express the association. Associated classes are stored under the class with which they are associated. The association class is not stored. This mechanism is well suited for weak associations because the superior/inferior relationship is well established.
7. Use DIT containment to express the association. Associated classes are stored under the class with which they are associated. The association class is not stored. Attributes from the association class are stored as part of the contained class. They are added to the contained class with an auxiliary class.
8. Use DIT containment to express the association. The association class is stored under one of the associated classes. Remaining associated classes are stored under the association class. If the association class has attributes, they are stored with the association class. This mechanism is useful for associations with cardinality greater than two.
9. De-normalize the association by combining the attributes of the associated classes. A new structural class is defined that has the combined attribute set. It is used in place of the individual members of the association.
10. De-normalize the association by combining the attributes of the associated classes. Define one member of the association as an auxiliary class so its

attributes may be added to the other member. This mechanism only works if the cardinality of the side of the association represented by an auxiliary class is one, or it only has one attribute

11. De-normalize the association by combining attributes of the associated classes. All of the attributes of one of the associated classes are combined into a single formatted string. An attribute for the string is added to the other class in the association. It is either single valued or multi-valued depending on the cardinality of the association.

Note:

References may be implemented using any of the methods discussed above.

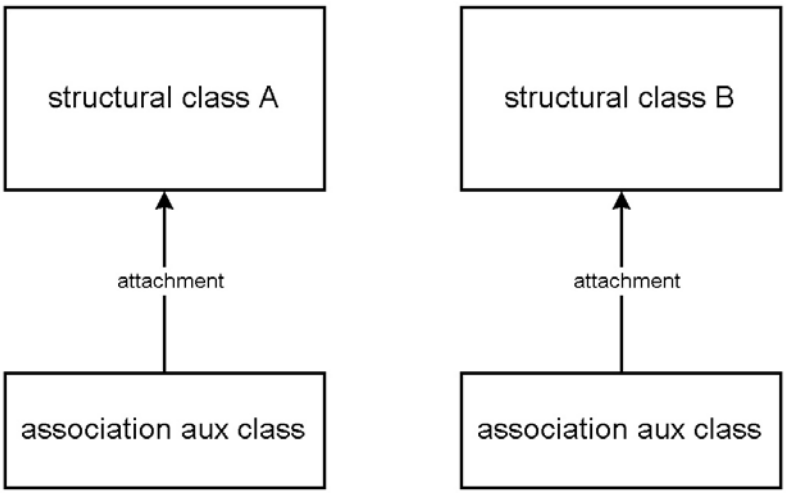
3.5 Recommended association mapping mechanism

In many situations, the choice between mechanisms for mapping associations is somewhat arbitrary. However, to improve consistency and interoperability it is desirable that, when possible, all mappings use the same approach. This section describes the association mapping used by the DMTF for the core schema, and provides it as a model for use by other mappings.

The core schema mapping uses a combination of three mechanisms discussed previously. They are:

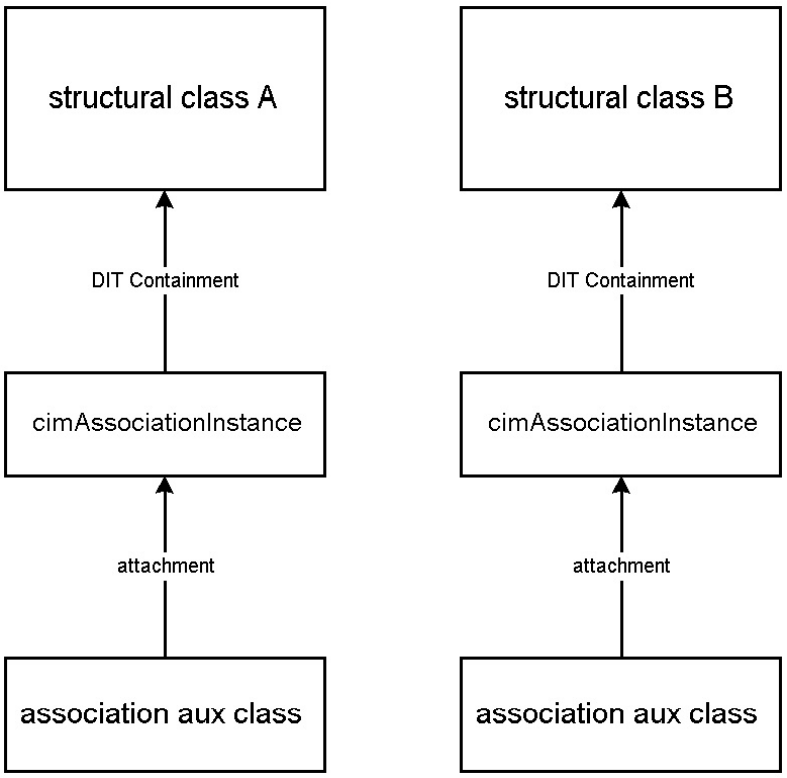
- Adding reference attributes to each member of the association through auxiliary classes
- Adding references to each member of the association through DIT containment of a helper class containing the references
- Direct DIT containment for weak associations

The first two mechanisms are related. In the core mapping, all non-weak associations classes are defined as auxiliary classes which have references to both members of the association. The auxiliary classes are attached to each member of the association so they are mutually referencing.



In some cases, it may not be practical to attach the association directly to associated classes. This may be the case if an entry participates in multiple instances of an

association, or if the directory server restricts the behavior of auxiliary classes. In these cases, the references may be stored under each member of the association. For the sake of simplicity, both sides of an association are always mapped the same way. That is, either both sides use attached auxiliary classes or both sides use DIT containment of the references.



To avoid the need to declare new structural classes for the DIT-contained references, a single helper structural class, cimAssociationInstance, is used for all such mappings. The auxiliary class that would have been attached to the associated class is instead attached to the helper class. Therefore, cimAssociationInstance may have any CIM association auxiliary class attached to it This approach allows a single schema definition to support either approach. The helper class is defined as:

cimAssociationInstance

```
(
      1.3.6.1.4.1.412.100.1.1.1.1
      NAME        'cimAssociationInstance'
      SUP         top
      MUST        (cimAssociationName)
      MAY         (cimAssociationTypeName)
)
```

```
cimAssociationName
(
      1.3.6.1.4.1.412.100.1.2.3
      NAME        'cimAssociationName'
      DESC        'The RDN of this association instance'
      SYNTAX      DirectoryString
      SINGLE-VALUE
)
cimAssociationTypeName
(
      1.3.6.1.4.1.412.100.1.2.4
      NAME        'cimAssociationTypeName'
      DESC        'support storing extra information about the
                  association type'
      SYNTAX      DirectoryString
      SINGLE-VALUE
)

(
      <sr5>
      NAME        'cimAssociationInstanceStructureRule'
      FORM        cimAssociationInstanceNameForm
      SUP         (<sr1> <sr2> <sr3> <sr4>)
 )
```

The attribute cimAssociationName is used to construct the RDN of entries of class cimAssociationInstance. To permit directory implementers to use implementation-dependent values for cimAssociationName, cimAssociationTypeName is also defined. It is intended to hold the CIM Association Type (e.g. "CIM_VLANFor") to aid when searching.

4 Data Type Mapping

There are several inconsistencies between the data types supported by CIM and the syntaxes supported by directories. This section proposes mechanisms to map CIM data types into directories.

4.1 Character set

The vast majority of directory syntaxes are string based. Because of this the character set or code page used in the string may be of issue. CIM specifies USC-2 as the character set for its strings. RFC 2251[2] specifies UTF-8 as the character set to use with the LDAP protocol. Other character sets may be used by non-conforming implementations. In most situations, the selected programming environment provides a consistent character set; however, some applications may need to perform character set

translation. When such translation is required, it should be done in accordance with RFC 2279[3].

In the mappings described below all characters used have ASCII values of 127 or less which means that they have the same code point in all character sets in wide usage.

4.2 DateTime Mapping

The CIM DateTime type is used to store moments in time, and time intervals. When used to store moments in time, the semantics are the same as the Generalized Time syntax specified for directories in X.208[4], and the syntax is similar. DateTime properties that store moments may be mapped to Generalized Time. This assumes that the semantics of a property is clear. That is it will never contain an interval value.

4.2.1 Mapping Moments

Both CIM DateTime and Directory Generalized Time are used to store a date and a time combination as a string. However, there are several syntactic differences in the formats. They are:

- The character set used (see previous discussion).
- Reduced accuracy is represented differently.
- DateTime is a fixed length string Generalized Time is a variable length format.
- DateTime uses hundreds of minutes ("mmm") to specify offset from UTC. Generalized Time uses hours and minutes ("hhmm").
- DateTime uses zero minutes offset to specify UTC. Generalized Times uses a “Z”.
- Generalized Time allows for either a period “.” or a comma “,” to be used as the decimal separator. DateTime requires a period.

Accuracy

Both CIM DateTime and directory Generalized Time provides for specifying times with reduced precision. However, the mechanism is not the same. CIM DateTime is a fixed length format so fields that are not significant must be replaced with a placeholder. The “*” asterisk is used. Generalized Time is variable length syntax. Non significant fields may be omitted starting form the right up to the entire time portion of the string. The entire date must be present.

When mapping from Generalized Time to DateTime, all omitted fields in the Generalized Time value must be present in the DateTime value and filled with asterisks.

When mapping from DateTime to Generalized Time, all contiguous asterisk-filled fields starting from the right most (microseconds) up to the first date field, may be omitted. Any remaining asterisk-filled fields must be zero filled. Note: the semantics of non-significant fields embedded in a DateTime value is unclear.

Mapping algorithms

DateTime to Generalized Time:

1. Perform character set translation as required.

2. Map asterisks as described above.
3. If the UTC offset is +000 it is replaced with a “Z”. Otherwise the UTC offset is translated from minutes to hours and minutes format.

Generalized Time to DateTime:

1. Zero pad or truncate the decimal portion of the seconds to be exactly six digits. If there are no decimal seconds specified then use the decimal point “.” and six asterisks.
2. If the value is in UTC, that is, it is followed by a “Z” the “Z” is replaced with +000. Otherwise, the UTC offset is translated from hours and minutes(“hhmm”) format to minutes(“mmm”) format.
3. If a comma is used as the decimal separator, replace it with a period.
4. Perform character set translation as required.

4.3 Real Mapping

CIM supports IEEE four byte and eight byte floating point values as real32 and real64 types. However, the LDAP protocol has no specific support for floating point values. Because LDAP is a string-based protocol, any binary or byte coded representation of reals in the directory raises byte ordering issues. To avoid these, and to remain in keeping with the spirit of the protocol, the recommended mapping is string based.

The mapping is intended to create a string representation of floating point numbers which allows standard string comparison functions to sort the values into their correct collating sequence based on their floating point value.

Because string comparison is positional, it is necessary to define a fix format for representing the mantissa, and the exponent. Because the collating sequence for string comparison is left to right, the most significant portion of the representation must be on the left. There are four separate cases that must be handled.

- Negative mantissa and positive exponent
- Negative mantissa and negative exponent
- Positive mantissa and negative exponent
- Positive mantissa, and positive exponent

The above list is ordered by the desired collating sequence from smallest value to largest value. A single representation does not provide the correct collating sequence for all cases. Therefore, it is necessary to sort by case, and then to sort within each case. To accomplish this, the cases are numbered from one to five as follows:

1. Negative mantissa and positive exponent
2. Negative mantissa and negative exponent
3. Zero
4. Positive mantissa and negative exponent
5. Positive mantissa, and positive exponent

For symmetry, zero is treated as its own case instead of a special sub-case of case 4.

4.3.1 String format

A 64-bit float has a range of 1.7976931348623158e+308 to 2.2250738585072014e-308[5]. To represent this as a string, three digits are required for the exponent, and 17 for the mantissa, not including the decimal point. The directory representation is fixed format, zero padded, blank separated, with the most significant fields on the left. The first character in the string is the case number. For readability, it is followed by a blank. Next is a three digit exponent, again followed by a blank. Next are a single digit, a decimal point, and 16 digits of decimal.

		Exp				16 digits zero padded to right															
5		n	n	n		n	.	n	n	n	n	n	n	n	n	n	n	n	n	n	n

The way each of the fields is interpreted varies with the case.

The cases are examined in reverse order so the simplest may be examined first.

Positive mantissa, and positive exponent (case 5)

This is relatively straightforward. The exponent field has the exponent value expressed as a three-digit integer string. It is zero padded to the left if necessary. The mantissa field as a seventeen-digit decimal string with exactly one digit to the left of the decimal point, for a total of 18 characters. It is zero padded to the right if necessary.

Notes:

- The first digit is a 5 to indicate the case.
- There is exactly one digit to the left of the decimal place. It is always non zero.
- Positions 2 through 4 have the exponent. It is right justified, and zero padded to the left if it is less than three digits.
- Spaces are added to aid human readability.
- No signs are required for the exponent or the mantissa because they are expressed in the case number.

Positive mantissa and negative exponent (case 4)

When the exponent is negative, larger whole number values for the exponent produce smaller actual values. For this case, a string comparison of the numeric representation of the exponent yields the reverse of the desired collating sequence. To flip the collating sequence, the value of the exponent is added to 999, and the result stored as the exponent. No sign is stored. The sign of both the exponent and mantissa are indicated by the case character.

Zero (case 3)

The case number is sufficient to insure the correct collating sequence. To insure equality comparisons work correctly, all remaining digits are zero.

Negative mantissa and negative exponent (case 2)

When both the exponent and the mantissa are negative, the collating order for the exponent is correct. A larger exponent yields a number that is closer to zero and therefore larger. However, the collating sequence for the mantissa is reversed. To flip the collating sequence for the mantissa it is added to 10, and the result stored.

Negative mantissa and positive exponent (case 1)

When the mantissa is negative and the exponent is positive, the collating sequence is flipped for both of them. This is achieved by subtracting the exponent to 999, and adding the mantissa to 10.

4.3.2 Examples

Value	Representation
3.25e5	5 005 3.2500000000000000
8.4e-5	4 994 8.4000000000000000
8.4e-7	4 992 8.4000000000000000
7.23e-7	4 992 7.2300000000000000
0.0e0	3 000 0.0000000000000000
-4.25e-4	2 004 5.7500000000000000
-6.35e-4	2 004 3.6500000000000000
-6.35e-3	2 003 3.6500000000000000
-4.0e104	1 895 6.0000000000000000
-4.0e105	1 894 6.0000000000000000
-6.0e105	1 894 4.0000000000000000

4.3.3 32-bit vs. 64-bit values

CIM supports both 32- and 64-bit floating-point values. To allow comparisons between the two, both are stored in the 64-bit format described above. This implies a greater degree of precision than is actually available for 32-bit values. This is considered acceptable. The directory mapping described below provides implicit documentation of the actual precision of a value.

4.3.4 Directory mapping

The intent of the mapping is to simulate a new syntax. To foster that illusion, and to aid in documentation, two new attributes are defined. cimFloat32, and CimFloat64. All CIM floating point attributes will be derived from either cimFloat32 or cimFloat64 as appropriate. They are defined as:

```
CimFloat32
(
      1.3.6.1.4.1.412.100.1.2.6
      NAME        'cimFloat32'
      DESC        'CIM 32 bit float encoded as CIM sortable float
                  format'
      EQUALITY    caseIgnoreMatch
      ORDERING    caseIgnoreOrderingMatch
      SYNTAX      1.3.6.1.4.1.1466.115.121.1.15
)

CimFloat64
(
      1.3.6.1.4.1.412.100.1.2.7
      NAME        'cimFloat64'
      DESC        'CIM 64 bit float encoded as CIM sortable float
                  format'
      EQUALITY    caseIgnoreMatch
      ORDERING    caseIgnoreOrderingMatch
      SYNTAX      1.3.6.1.4.1.1466.115.121.1.15
)
```

5 Other Mapping Issues

5.1 Indexed Arrays

Because LDAP multi-valued attributes do not provide ordering, there is no easy mechanism for mapping CIM index arrays to directories. Possible solutions include:

1. Defining a new syntax, and possible matching rules.

   Since most commercial directories are not true X.500 implementations, adding a new syntax requires updating the directory server. Although directory servers may add support for index arrays in the future, such a change is beyond the scope of this paper.

2. Define a formatted string that contains the indexing information, and is manipulated by the client. This could be done by defining a single string that contains the entire array, or by using a multi-valued attribute and adding an index tag to each attribute value.

   This requires extra work on the part of the client, requires the client to understand the format, and requires the cooperation of all clients who update the data to insure it is correct.

   Access and update performance may be a problem, especially for a large array

3. Define a new structural class to replace the indexed property. The class contains an index attribute, which is the RDN, and the attribute that is the array property. There is an instance of the class for each array element. Instances are stored under the entry that contains the array.

   Again, clients must cooperate to insure the integrity of the data.

   If there are a large number of arrays, or the arrays are large, the additional entries could degrade the performance of the directory server.

4. Redefine the schema to eliminate the need for the indexing. For example, if two parallel indexed arrays are used to relate values with the same index, a new class can be defined that has both properties. For each index an entry is created, that has the values from each array. Entries of the new class are stored under the entry, which contained the arrays.

   If there are a large number of arrays, or the arrays are large, the additional entries could degrade the performance of the directory server.

   Note:

   The DMTF core schema mapping uses this technique to eliminate parallel arrays. It is combined with approach 3 to retain the index value.

5.2 DIT Structure

It is neither reasonable nor desirable for a mapping to specify or constrain the overall structure of the Directory Tree. This is the true because the semantics of the directory tree are already overloaded. The structure of the directory tree is used to determine access control for entries, and partition the tree across physical servers.

However, it is desirable to have a well-known structure for specific groups of related classes. This is especially true when DIT containment is used to map associations.

To facilitate this, some classes can be designated as top-level container classes. The remaining mapped classes may be constrained in the placement to be placed either directly or indirectly below the top-level classes. It is desirable that the classes selected to be top level are course enough grained that they have meaning to the directory at large.

5.3 Versioning

Object/entry definitions are expected to change between versions of CIM schema; therefore, mapping techniques should recognize this and provide mechanisms to distinguish between object versions. Because there is no explicit support for schema versioning in directories, it is not possible to provide a mechanism that guarantees application compatibility across versions. However, it is possible to provide naming conventions and guidelines that reduce unnecessary incompatibility, and allows an application to detect when they exist. They are as follows:

- The definition of some classes may change between schema versions. Therefore, the schema entry name for CIM classes contains the schema version from which they were mapped.
- Attribute definitions are considered much less likely to change. Therefore, their name does not contain a version number.

- New schema entries are only created for class definitions in new CIM schema versions that have changed in ways that materially effect the mapping. This includes anything that changes the ASN.1 string, which is the schema entry. This includes addition or removal of any properties or the change of a class's superior. A class may have a new superior because of a change that forced its superior to move to a new version. Thus, changes may cascade down the inheritance hierarchy.

  This implies that a given version of a schema may have classes with names from any or all previously mapped versions.

5.4 Naming

Mapping CIM schema requires the specification of many new directory schema entries. Each of these entries requires a name. As much as possible, the name of a mapped directory schema element should be derived from its CIM name and vise versa. This is not possible in cases where there are name collisions.

Naming guidelines

Following are guidelines for naming new directory schema elements.

- Where there is a match in both semantics and type, it is desirable to reuse attributes defined in RFC 2256.
- When two or more classes have a property with identical name and type, a single attribute should be declared and shared among the classes.
- When two or more classes have properties with the same name, but different types, the name of all but one of the properties must be mangled. The un-mangled name should be used for the property of the class highest (closest to the root) in the inheritance hierarchy.

  Future versions of this document may specify a name-mangling scheme.
- OIDs for directory objects that are required by mappings are standardized by the DMTF, and assigned by the DMTF.

Name Structure

Where it is necessary to create new attributes or classes, their names should be prefixed with "cim". Class names have the form:

cimVV*ClassName*

Where ClassName is the name of the CIM class, and VV is the version of the schema in which the mapped definition first appeared in its current form. Class names contain version tags because they are expected to change from time to time.

Attributes names have the form:

cim*PropertyName*

where PropertyName is the name of a property belonging to a mapped CIM class. Attribute names do not have version tags because they are expected to change only rarely. In the rare instance when a CIM property definition changes significantly enough

to require a change to the mapped attribute, but doesn't change its name, a new attribute must be created with a mangled name.

Naming de-normalized directory classes

When the mapping de-normalizes the CIM schema, more than one CIM class is mapped to a single directory class. In these cases, there are several options for naming the directory class. They include

- Selecting the name of one of the mapped classes for the directory class name. This is appropriate if all but one of the classes are associations. Names of association can generally be dropped because they are covered by the association mapping selected. It may also be appropriate when weak associations are de-normalized. The name of the primary class may be used.
- Add a level of inheritance to the directory class definition for each addition CIM classes mapped to it. This allows the value of the objectless attribute to have the name of all mapped CIM classes.
- Concatenate the names of all mapped CIM classes together to form the directory class name.

5.5 OID Structure

Although the OID is considered an opaque object, it does have internal structure. The following guidelines are suggested to aid human readability to OIDs assigned to mapped schema elements.

5.5.1 Guidelines

The OID should be structured:

BaseOID.schema.entity_type.version.entity_id

Where:

BaseOID:	Is the root of the OID sub tree of the assigning organization
Schema:	is the identity of the CIM schema being mapped. It is arbitrary, but must be unique. For schemas with OIDs assigned from the DMTF tree, the schema ID must be obtained from the DMTF.
Entity_type:	identifies the type of directory entity where: 1 = Object class 2 = Attribute 3 = Name form
Version:	is the version of the specific entity. Versions start at one and are incremented each time a new version of an entity is needed.
Entity_Id:	identifies the specific entity. Initial assignment is arbitrary, but once an ID is assigned, it is constant through all versions of the entity.

5.5.2 Example

The CIM 2.2 class System is mapped to a directory class called cim22System. The OID is assigned from the DMTF subtree as follows.

1.3.6.1.4.1.412.100.2.1.1.17

- 1.3.6.1.4.1.412.100 is the OID subtree assigned by the DMTF for directory schema mapping.
- 2 is a the branch allocated for mapping the core schema.
- 1 indicates that the mapping is an object class.
- 1 indicates that it is the first version mapped.
- 17 is the ID assigned to the System object class.

Assume in CIM 2.3 that the System class is altered in a way that requires a new version of the directory mapping. It is now mapped to the directory class cim23System that is assigned OID

```
1.3.6.1.4.1.412.100.2.1.2.17
```

The only change is that the version number is incremented.

Further, assume that after the cim23System definition is published, an error is discovered that requires issuing an updated mapping cim23aSystem. It is assigned OID

```
1.3.6.1.4.1.412.100.2.1.3.17
```

Notice that there may not be a correlation between the OID version arc and a CIM schema version.

5.5.3 DMTF OID Assignment

The DMTF has been assigned the OID:

```
1.3.6.1.4.1.412
```

and is responsible for the assignment of OIDs subordinate to it. The DMTF has assigned the OID

```
1.3.6.1.4.1.412.100
```

as a prefix for LDAP mappings of CIM schema done by the DMTF. In addition, the following schema specific assignments have been made.

```
1.3.6.1.4.1.412.100.1 General schema entities that support
                      schema mapping but are not specific to
                      a schema
1.3.6.1.4.1.412.100.2 the Core schema
1.3.6.1.4.1.412.100.3 the Physical schema
```

Other DMTF-sponsored mappings will be assigned a root OID directly below `1.3.6.1.4.1.412.100`

5.6 Reusing existing directory schema definitions

They are situations in which CIM classes are closely related to existing directory classes, or a CIM property has the same semantics as a widely used directory attribute. In these

cases, it is desirable to reuse the existing directory definitions in the mapping. This may occur because the CIM schema was originally derived from a directory schema, or because CIM and directory schemas have been separately developed to model the same domain. In the first case, it is likely that the two schemas are similar and a mapping can be made with relatively minor extensions to the directory schema. In the second case, a description of how to translate between the two schemas may be the most that can be provided.

5.6.1 Reusing Classes

There are several issues that must be addressed when existing directory classes are reused. These include:

- Class name
- RDN attribute

Class name

Because the guidelines provide for the inclusion of the CIM class name in the directory class name, some application may be written to search the objectClass attribute for CIM class names (for example, using a substring match). Existing directory classes must be extended if this capability is to be supported.

RDN attribute

Some directories specify the RDN for a class either as part of the class definition or as a name form rule. In these directories, it may not be possible to use orderedCimKeys or orderedCimModelPath for the RDN of the reused class. In addition, changing the RDN may hinder interoperability with existing applications.

There are three general approaches that can be used when it is necessary to reconcile minor differences in order to reuse existing classes. They are:

- Reuse the class unmodified,
- Extend the class through inheritance, or
- Extend the class by attaching an auxiliary class.

Each approach implies tradeoffs between interoperability with applications geared towards a CIM centric mapping, and application geared towards other directory schema.

Unmodified

A CIM class may be mapped to an unmodified directory class. It is likely that some CIM properties will not be mapped. This is the most directory centric option and provides the greatest interoperability with existing directory applications.

Inheritance

The additional attributes required by the CIM side of the mapping may be added to an existing entry by deriving a new class from the class to be reused.

Some directories allow the RDN for a class to be changed by derived classes. For these directories, inheritance is the preferred approach for applications that wish to use orderedCimKeys or orderedCimModelPath for all RDNs. A problem with this technique is that it may introduce problems for existing directory applications that assume a strict extension semantics in subclassing of the derived classes. That is, if the interpretation of the object-oriented semantics that an instance of a subclass is a valid instance of all of its superclasses is assumed to include the classes RDN attribute.

Another drawback to using inheritance is there may be other classes that inherit from the class being reused. The mapping can not be applied to these derived classes.

Despite these drawbacks, this approach may be useful for mappings who's primary concern is compatibility with other mappings conforming to these guideline, but must also provide some level of interoperability with existing schema.

Auxiliary Class

An auxiliary class can be defined for additional attributes defined by the CIM side of the mapping. This allows the mapping to be applied to all entries that have the target class anywhere in its ancestry. The auxiliary class should be named in conformance with section 5.4.

Some directories do not include the names of auxiliary classes in the value of the objectClass attribute. For these directories, applications will have problems if they make use of the CIM class name.

Many directories do not allow an entry's RDN to be specified as an attribute supplied by an auxiliary class. In these directories, this approach prohibits the use of CIM specific RDNs.

5.6.2 Reusing Attributes

There is much less flexibility in the handling of attributes then there is in the handling of classes. It is not possible to access an attribute by the name of one of its ancestors, so it is not possible to add the name of the mapped CIM property to attributes the way it can be added to classes.

When existing classes are used by a mapping, all the attributes included in those classes are also used. The mapping from CIM property to attribute is implied, and must be understood by client applications. This may cause applications that rely on the naming scheme defined in section 5.4 problems.

Because some directory applications are attribute based as opposed to class based, it may be desirable to reuse individual standard attributes outside the context of existing classes. In such cases, the mapper should insure that there is a good fit between the semantics of the CIM property and the attribute. This should take into account the implied semantics of the general usage of the attribute.

5.7 Must vs. May Attributes

It is recommended that all attributes be mapped as MAY attributes. It is the responsibility of a specific application to insure the properties it needs are mapped and present. If a class mapping is derived from existing non CIM classes either by inheritance or via auxiliary classes, then any MUST attributes in the existing class(es) need to be accommodated.

RDN

Since two options are provided for the RDN attribute, neither is specified as a Must attribute. Specific installations may elevate the selected RDN attribute to Must.

CIM Keys and Propagated Keys

It would normally be desirable to insure that all key attributes are present by mapping properties with the Key qualifier to Must attributes. However, both options for the RDN attribute require that the values for all key properties be present in the RDN. Requiring Key properties to be present separately would force a duplication of data that may not be desirable.

Likewise, the value of propagated keys is available from either the RDN value or the DN value. To avoid data duplication, it is recommended that propagated keys not be mapped in the weak side of the association at all.

6 Appendix 1 – Schema Elements

6.1 Classes

6.1.1 cimAssociationInstance

```
(
        1.3.6.1.4.1.412.100.1.1.1.1
        NAME        'cimAssociationInstance'
        SUP         top
        MUST        (cimAssociationName)
        MAY         (cimAssociationTypeName)
)
```

6.2 Attributes

6.2.1 orderedCimKeys

```
(
        1.3.6.1.4.1.412.100.1.2.1
        NAME        'orderedCimKeys'
        DESC        'The model path for the instance (without
                    propagated keys). May be used as an RDN'
        SYNTAX      DirectoryString
        SINGLE-VALUE
        EQUALITY    octetStringMatch
)
```

6.2.2 orderedCimModelPath

```
(
        1.3.6.1.4.1.412.100.1.2.2
        NAME        'orderedCimModelPath'
        DESC        'The model path for the instance (with propagated
                    keys). May be used as an RDN'
        SYNTAX       DirectoryString
        SINGLE-VALUE
        EQUALITY    octetStringMatch
)
```

6.2.3 cimAssociationName

```
(
        1.3.6.1.4.1.412.100.1.2.3
        NAME        'cimAssociationName'
        DESC        'The associations" name'
        SYNTAX      DirectoryString
        SINGLE-VALUE
)
```

6.2.4 cimAssociationTypeName

```
(
        1.3.6.1.4.1.412.100.1.2.4
        NAME        'cimAssociationTypeName'
        DESC        'support storing extra information about the
                    association type'
        SYNTAX      DirectoryString
        SINGLE-VALUE
)
```

6.2.5 arrayIndex

```
(
        1.3.6.1.4.1.412.100.1.2.5
        NAME        'arrayIndex'
        DESC        'the index of this child'
        SYNTAX      integer
        EQUALITY    integerMatch
)
```

6.2.6 cimFloat32

```
 (
        1.3.6.1.4.1.412.100.1.2.6
        NAME        ‘CimFloat32’
        DESC        ’CIM 32 bit float encoded as CIM sortable float
                    format’
        EQUALITY    caseIgnoreeMatch
        ORDERING    caseIgnoreOrderingMatch
        SYNTAX      1.3.6.1.4.1.1466.115.121.1.15
)
```

6.2.7 cimFloat64

```
(
        1.3.6.1.4.1.412.100.1.2.7
        NAME        'CimFloat64'
        DESC        'CIM 64 bit float encoded as CIM sortable float
                    format'
        EQUALITY    caseIgnoreeMatch
        ORDERING    caseIgnoreOrderingMatch
        SYNTAX      1.3.6.1.4.1.1466.115.121.1.15
)
```

6.3 Name Forms

6.3.1cimAssociationInstanceNameForm

```
(
        1.3.6.1.4.1.412.100.1.3.1.1
        NAME        'cimAssociationInstanceNameForm'
        OC          cimAssociationInstance
        MUST        (cimAssociationName)
)
```

---

[1] COMMON NFORMATION MODEL (CIM) SPECIFICATION VERSION 2.2 - DMTF

[2] M. Wahl, T. Howes, S. Kille, "Lightweight Directory Access Protocol (v3)," RFC 2251, December 1997

[3] F. Yergeau, "UTF-8, a transformation format of ISO 10646," RFC 2279, January 1998.

[4] OPEN SYSTEMS INTERCONNECTION MODEL AND NOTATIONSPECIFICATION OF ABSTRACT SYNTAX NOTATION ONE (ASN.1)

[5] From sys/limits.h on AIX 4.3