Network Working Group
Request for Comments: 3687
Category: Standards Track

S. Legg
Adacel Technologies

February 2004

Lightweight Directory Access Protocol (LDAP) and X.500 Component Matching Rules

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

The syntaxes of attributes in a Lightweight Directory Access Protocol (LDAP) or X.500 directory range from simple data types, such as text string, integer, or boolean, to complex structured data types, such as the syntaxes of the directory schema operational attributes. Matching rules defined for the complex syntaxes usually only provide the most immediately useful matching capability. This document defines generic matching rules that can match any user selected component parts in an attribute value of any arbitrarily complex attribute syntax.

Table of Contents

1. Introduction

The structure or data type of data held in an attribute of a Lightweight Directory Access Protocol (LDAP) [7] or X.500 [19] directory is described by the attribute's syntax. Attribute syntaxes range from simple data types, such as text string, integer, or boolean, to complex data types, for example, the syntaxes of the directory schema operational attributes.

In X.500, the attribute syntaxes are explicitly described by Abstract Syntax Notation One (ASN.1) [13] type definitions. ASN.1 type notation has a number of simple data types (e.g., PrintableString, INTEGER, BOOLEAN), and combining types (i.e., SET, SEQUENCE, SET OF, SEQUENCE OF, and CHOICE) for constructing arbitrarily complex data types from simpler component types. In LDAP, the attribute syntaxes are usually described in Augmented Backus-Naur Form (ABNF) [2], though there is an implied association between the LDAP attribute syntaxes and the X.500 ASN.1 types. To a large extent, the data types of attribute values in either an LDAP or X.500 directory are described by ASN.1 types. This formal description can be exploited to identify component parts of an attribute value for a variety of purposes. This document addresses attribute value matching.

With any complex attribute syntax there is normally a requirement to partially match an attribute value of that syntax by matching only selected components of the value. Typically, matching rules specific to the attribute syntax are defined to fill this need. These highly specific matching rules usually only provide the most immediately useful matching capability. Some complex attribute syntaxes don't even have an equality matching rule let alone any additional matching rules for partial matching. This document defines a generic way of matching user selected components in an attribute value of any arbitrarily complex attribute syntax, where that syntax is described using ASN.1 type notation. All of the type notations defined in X.680 [13] are supported.

Section 3 describes the ComponentAssertion, a testable assertion about the value of a component of an attribute value of any complex syntax.

Section 4 introduces the ComponentFilter assertion, which is an expression of ComponentAssertions. The ComponentFilter enables more powerful filter matching of components in an attribute value.

Section 5 defines the componentFilterMatch matching rule, which enables a ComponentFilter to be evaluated against attribute values.

Section 6 defines matching rules for component-wise equality matching of attribute values of any syntax described by an ASN.1 type definition.

Examples showing the usage of componentFilterMatch are in Section 7.

For a new attribute syntax, the Generic String Encoding Rules [9] and the specifications in sections 3 to 6 of this document make it possible to fully and precisely define the LDAP-specific encoding, the LDAP and X.500 binary encoding (and possibly other ASN.1 encodings in the future), a suitable equality matching rule, and a comprehensive collection of component matching capabilities, by simply writing down an ASN.1 type definition for the syntax. These implicit definitions are also automatically extended if the ASN.1 type is later extended. The algorithmic relationship between the ASN.1 type definition, the various encodings and the component matching behaviour makes directory server implementation support for the component matching rules amenable to automatic code generation from ASN.1 type definitions.

Schema designers have the choice of storing related items of data as a single attribute value of a complex syntax in some entry, or as a subordinate entry where the related data items are stored as separate attribute values of simpler syntaxes. The inability to search component parts of a complex syntax has been used as an argument for favouring the subordinate entries approach. The component matching rules provide the analogous matching capability on an attribute value of a complex syntax that a search filter has on a subordinate entry.

Most LDAP syntaxes have corresponding ASN.1 type definitions, though they are usually not reproduced or referenced alongside the formal definition of the LDAP syntax. Syntaxes defined with only a character string encoding, i.e., without an explicit or implied corresponding ASN.1 type definition, cannot use the component matching capabilities described in this document unless and until a semantically equivalent ASN.1 type definition is defined for them.

2. Conventions

Throughout this document "type" shall be taken to mean an ASN.1 type unless explicitly qualified as an attribute type, and "value" shall be taken to mean an ASN.1 value unless explicitly qualified as an attribute value.

Note that "ASN.1 value" does not mean a Basic Encoding Rules (BER) [17] encoded value. The ASN.1 value is an abstract concept that is independent of any particular encoding. BER is just one possible encoding of an ASN.1 value. The component matching rules operate at the abstract level without regard for the possible encodings of a value.

Attribute type and matching rule definitions in this document are provided in both the X.500 [10] and LDAP [4] description formats. Note that the LDAP descriptions have been rendered with additional white-space and line breaks for the sake of readability.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED" and "MAY" in this document are to be interpreted as described in BCP 14, RFC 2119 [1]. The key word "OPTIONAL" is exclusively used with its ASN.1 meaning.

3. ComponentAssertion

A ComponentAssertion is an assertion about the presence, or values of, components within an ASN.1 value, i.e., an instance of an ASN.1 type. The ASN.1 value is typically an attribute value, where the ASN.1 type is the syntax of the attribute. However, a ComponentAssertion may also be applied to a component part of an attribute value. The assertion evaluates to either TRUE, FALSE or Undefined for each tested ASN.1 value.

A ComponentAssertion is described by the following ASN.1 type (assumed to be defined with "EXPLICIT TAGS" in force):

```
ComponentAssertion ::= SEQUENCE {
    component         ComponentReference (SIZE(1..MAX)) OPTIONAL,
    useDefaultValues  BOOLEAN DEFAULT TRUE,
    rule              MATCHING-RULE.&id,
    value             MATCHING-RULE.&AssertionType }

ComponentReference ::= UTF8String
```

MATCHING-RULE.&id equates to the OBJECT IDENTIFIER of a matching rule. MATCHING-RULE.&AssertionType is an open type (formerly known as the ANY type).

The "component" field of a ComponentAssertion identifies which component part of a value of some ASN.1 type is to be tested, the "useDefaultValues" field indicates whether DEFAULT values are to be substituted for absent component values, the "rule" field indicates

how the component is to be tested, and the "value" field is an asserted ASN.1 value against which the component is tested. The ASN.1 type of the asserted value is determined by the chosen rule.

The fields of a ComponentAssertion are described in detail in the following sections.

3.1. Component Reference

The component field in a ComponentAssertion is a UTF-8 character string [6] whose textual content is a component reference, identifying a component part of some ASN.1 type or value. A component reference conforms to the following ABNF [2], which extends the notation defined in Clause 14 of X.680 [13]:

```
component-reference = ComponentId *( "." ComponentId )
ComponentId         = identifier /
                      from-beginning /
                      count /
                      from-end /       ; extends Clause 14
                      content /        ; extends Clause 14
                      select /         ; extends Clause 14
                      all

identifier          = lowercase *alphanumeric
                         *(hyphen 1*alphanumeric)
alphanumeric        = uppercase / lowercase / decimal-digit
uppercase           = %x41-5A  ; "A" to "Z"
lowercase           = %x61-7A  ; "a" to "z"
hyphen              = "-"

from-beginning      = positive-number
count               = "0"
from-end            = "-" positive-number
content             = %x63.6F.6E.74.65.6E.74 ; "content"
select              = "(" Value *( "," Value ) ")"
all                 = "*"


positive-number     = non-zero-digit *decimal-digit

decimal-digit       = %x30-39  ; "0" to "9"
non-zero-digit      = %x31-39  ; "1" to "9"
```

An <identifier> conforms to the definition of an identifier in ASN.1 notation (Clause 11.3 of X.680 [13]). It begins with a lowercase letter and is followed by zero or more letters, digits, and hyphens. A hyphen is not permitted to be the last character and a hyphen is not permitted to be followed by another hyphen.

The <Value> rule is described by the Generic String Encoding Rules (GSER) [9].

A component reference is a sequence of one or more ComponentIds where each successive ComponentId identifies either an inner component at the next level of nesting of an ASN.1 combining type, i.e., SET, SEQUENCE, SET OF, SEQUENCE OF, or CHOICE, or a specific type within an ASN.1 open type.

A component reference is always considered in the context of a particular complex ASN.1 type. When applied to the ASN.1 type the component reference identifies a specific component type. When applied to a value of the ASN.1 type a component reference identifies zero, one or more component values of that component type. The component values are potentially in a DEFAULT value if useDefaultValues is TRUE. The specific component type identified by the component reference determines what matching rules are capable of being used to match the component values.

The component field in a ComponentAssertion may also be absent, in which case the identified component type is the ASN.1 type to which the ComponentAssertion is applied, and the identified component value is the whole ASN.1 value.

A valid component reference for a particular complex ASN.1 type is constructed by starting with the outermost combining type and repeatedly selecting one of the permissible forms of ComponentId to identify successively deeper nested components. A component reference MAY identify a component with a complex ASN.1 type, i.e., it is not required that the component type identified by a component reference be a simple ASN.1 type.

3.1.1. Component Type Substitutions

ASN.1 type notation has a number of constructs for referencing other defined types, and constructs that are irrelevant for matching purposes. These constructs are not represented in a component reference in any way and substitutions of the component type are performed to eliminate them from further consideration. These substitutions automatically occur prior to each ComponentId, whether constructing or interpreting a component reference, but do not occur after the last ComponentId, except as allowed by Section 3.2.

If the ASN.1 type is an ASN.1 type reference then the component type is taken to be the actual definition on the right hand side of the type assignment for the referenced type.

If the ASN.1 type is a tagged type then the component type is taken to be the type without the tag.

If the ASN.1 type is a constrained type (see X.680 [13] and X.682 [15] for the details of ASN.1 constraint notation) then the component type is taken to be the type without the constraint.

If the ASN.1 type is an ObjectClassFieldType (Clause 14 of X.681 [14]) that denotes a specific ASN.1 type (e.g., MATCHING-RULE.&id denotes the OBJECT IDENTIFIER type) then the component type is taken to be the denoted type. Section 3.1.6 describes the case where the ObjectClassFieldType denotes an open type.

If the ASN.1 type is a selection type other than one used in the list of components for a SET or SEQUENCE type then the component type is taken to be the selected alternative type from the named CHOICE.

If the ASN.1 type is a TypeFromObject (Clause 15 of X.681 [14]) then the component type is taken to be the denoted type.

If the ASN.1 type is a ValueSetFromObjects (Clause 15 of X.681 [14]) then the component type is taken to be the governing type of the denoted values.

3.1.2. Referencing SET, SEQUENCE and CHOICE Components

If the ASN.1 type is a SET or SEQUENCE type then the <identifier> form of ComponentId may be used to identify the component type within that SET or SEQUENCE having that identifier. If <identifier> references an OPTIONAL component type and that component is not present in a particular value then there are no corresponding component values. If <identifier> references a DEFAULT component type and useDefaultValues is TRUE (the default setting for useDefaultValues) and that component is not present in a particular value then the component value is taken to be the default value. If <identifier> references a DEFAULT component type and useDefaultValues is FALSE and that component is not present in a particular value then there are no corresponding component values.

If the ASN.1 type is a CHOICE type then the <identifier> form of ComponentId may be used to identify the alternative type within that CHOICE having that identifier. If <identifier> references an alternative other than the one used in a particular value then there are no corresponding component values.

The COMPONENTS OF notation in Clause 24 of X.680 [13] augments the defined list of components in a SET or SEQUENCE type by including all the components of another defined SET or SEQUENCE type respectively. These included components are referenced directly by identifier as though they were defined in-line in the SET or SEQUENCE type containing the COMPONENTS OF notation.

The SelectionType (Clause 29 of X.680 [13]), when used in the list of components for a SET or SEQUENCE type, includes a single component from a defined CHOICE type. This included component is referenced directly by identifier as though it was defined in-line in the SET or SEQUENCE type.

The REAL type is treated as though it is the SEQUENCE type defined in Clause 20.5 of X.680 [13].

The EMBEDDED PDV type is treated as though it is the SEQUENCE type defined in Clause 33.5 of X.680 [13].

The EXTERNAL type is treated as though it is the SEQUENCE type defined in Clause 8.18.1 of X.690 [17].

The unrestricted CHARACTER STRING type is treated as though it is the SEQUENCE type defined in Clause 40.5 of X.680 [13].

The INSTANCE OF type is treated as though it is the SEQUENCE type defined in Annex C of X.681 [14].

The <identifier> form MUST NOT be used on any other ASN.1 type.

3.1.3. Referencing SET OF and SEQUENCE OF Components

If the ASN.1 type is a SET OF or SEQUENCE OF type then the <from-beginning>, <from-end>, <count> and <all> forms of ComponentId may be used.

The <from-beginning> form of ComponentId may be used to identify one instance (i.e., value) of the component type of the SET OF or SEQUENCE OF type (e.g., if Foo ::= SET OF Bar, then Bar is the component type), where the instances are numbered from one upwards. If <from-beginning> references a higher numbered instance than the last instance in a particular value of the SET OF or SEQUENCE OF type then there is no corresponding component value.

The <from-end> form of ComponentId may be used to identify one instance of the component type of the SET OF or SEQUENCE OF type, where "-1" is the last instance, "-2" is the second last instance,

and so on. If <from-end> references a lower numbered instance than the first instance in a particular value of the SET OF or SEQUENCE OF type then there is no corresponding component value.

The <count> form of ComponentId identifies a notional count of the number of instances of the component type in a value of the SET OF or SEQUENCE OF type. This count is not explicitly represented but for matching purposes it has an assumed ASN.1 type of INTEGER (0..MAX). A ComponentId of the <count> form, if used, MUST be the last ComponentId in a component reference.

The <all> form of ComponentId may be used to simultaneously identify all instances of the component type of the SET OF or SEQUENCE OF type. It is through the <all> form that a component reference can identify more than one component value. However, if a particular value of the SET OF or SEQUENCE OF type is an empty list, then there are no corresponding component values.

Where multiple component values are identified, the remaining ComponentIds in the component reference, if any, can identify zero, one or more subcomponent values for each of the higher level component values.

The corresponding ASN.1 type for the <from-beginning>, <from-end>, and <all> forms of ComponentId is the component type of the SET OF or SEQUENCE OF type.

The <from-beginning>, <count>, <from-end> and <all> forms MUST NOT be used on ASN.1 types other than SET OF or SEQUENCE OF.

3.1.4. Referencing Components of Parameterized Types

A component reference cannot be formed for a parameterized type unless the type has been used with actual parameters, in which case the type is treated as though the DummyReferences [16] have been substituted with the actual parameters.

3.1.5. Component Referencing Example

Consider the following ASN.1 type definitions.

```
ExampleType ::= SEQUENCE {
    part1       [0] INTEGER,
    part2       [1] ExampleSet,
    part3       [2] SET OF OBJECT IDENTIFIER,
    part4       [3] ExampleChoice }
```

```
ExampleSet ::= SET {
    option      PrintableString,
    setting     BOOLEAN }

ExampleChoice ::= CHOICE {
    eeny-meeny  BIT STRING,
    miney-mo    OCTET STRING }
```

Following are component references constructed with respect to the type ExampleType.

The component reference "part1" identifies a component of a value of ExampleType having the ASN.1 tagged type [0] INTEGER.

The component reference "part2" identifies a component of a value of ExampleType having the ASN.1 type of [1] ExampleSet

The component reference "part2.option" identifies a component of a value of ExampleType having the ASN.1 type of PrintableString. A ComponentAssertion could also be applied to a value of ASN.1 type ExampleSet, in which case the component reference "option" would identify the same kind of information.

The component reference "part3" identifies a component of a value of ExampleType having the ASN.1 type of [2] SET OF OBJECT IDENTIFIER.

The component reference "part3.2" identifies the second instance of the part3 SET OF. The instance has the ASN.1 type of OBJECT IDENTIFIER.

The component reference "part3.0" identifies the count of the number of instances in the part3 SET OF. The count has the corresponding ASN.1 type of INTEGER (0..MAX).

The component reference "part3.*" identifies all the instances in the part3 SET OF. Each instance has the ASN.1 type of OBJECT IDENTIFIER.

The component reference "part4" identifies a component of a value of ExampleType having the ASN.1 type of [3] ExampleChoice.

The component reference "part4.miney-mo" identifies a component of a value of ExampleType having the ASN.1 type of OCTET STRING.

3.1.6. Referencing Components of Open Types

If a sequence of ComponentIds identifies an ObjectClassFieldType denoting an open type (e.g., ATTRIBUTE.&Type denotes an open type) then the ASN.1 type of the component varies. An open type is typically constrained by some other component(s) in an outer enclosing type, either formally through the use of a component relation constraint [15], or informally in the accompanying text, so the actual ASN.1 type of a value of the open type will generally be known. The constraint will also limit the range of permissible types. The <select> form of ComponentId may be used to identify one of these permissible types in an open type. Subcomponents of that type can then be identified with further ComponentIds.

The other components constraining the open type are termed the referenced components [15]. The <select> form contains a list of one or more values which take the place of the value(s) of the referenced component(s) to uniquely identify one of the permissible types of the open type.

Where the open type is constrained by a component relation constraint, there is a <Value> in the <select> form for each of the referenced components in the component relation constraint, appearing in the same order. The ASN.1 type of each of these values is the same as the ASN.1 type of the corresponding referenced component. The type of a referenced component is potentially any ASN.1 type however it is typically an OBJECT IDENTIFIER or INTEGER, which means that the <Value> in the <select> form of ComponentId will nearly always be an <ObjectIdentifierValue> or <IntegerValue> [9]. Furthermore, component relation constraints typically have only one referenced component.

Where the open type is not constrained by a component relation constraint, the specification introducing the syntax containing the open type should explicitly nominate the referenced components and their order, so that the <select> form can be used.

If an instance of <select> contains a value other than the value of the referenced component used in a particular value of the outer enclosing type then there are no corresponding component values for the open type.

3.1.6.1. Open Type Referencing Example

The ASN.1 type AttributeTypeAndValue [10] describes a single attribute value of a nominated attribute type.

```
   AttributeTypeAndValue ::= SEQUENCE {
       type    ATTRIBUTE.&id ({SupportedAttributes}),
       value   ATTRIBUTE.&Type ({SupportedAttributes}{@type}) }
```

ATTRIBUTE.&id denotes an OBJECT IDENTIFIER and ({SupportedAttributes}) constrains the OBJECT IDENTIFIER to be a supported attribute type.

ATTRIBUTE.&Type denotes an open type, in this case an attribute value, and ({SupportedAttributes}{@type}) is a component relation constraint that constrains the open type to be of the attribute syntax for the attribute type. The component relation constraint references only the "type" component, which has the ASN.1 type of OBJECT IDENTIFIER, thus if the <select> form of ComponentId is used to identify attribute values of specific attribute types it will contain a single OBJECT IDENTIFIER value.

The component reference "value" on AttributeTypeAndValue refers to the open type.

One of the X.500 standard attributes is facsimileTelephoneNumber [12], which is identified with the OBJECT IDENTIFIER 2.5.4.23, and is defined to have the following syntax.

```
   FacsimileTelephoneNumber ::= SEQUENCE {
       telephoneNumber PrintableString(SIZE(1..ub-telephone-number)),
       parameters      G3FacsimileNonBasicParameters OPTIONAL }
```

The component reference "value.(2.5.4.23)" on AttributeTypeAndValue specifies an attribute value with the FacsimileTelephoneNumber syntax.

The component reference "value.(2.5.4.23).telephoneNumber" on AttributeTypeAndValue identifies the telephoneNumber component of a facsimileTelephoneNumber attribute value. The component reference "value.(facsimileTelephoneNumber)" is equivalent to "value.(2.5.4.23)".

If the AttributeTypeAndValue ASN.1 value contains an attribute type other than facsimileTelephoneNumber then there are no corresponding component values for the component references "value.(2.5.4.23)" and "value.(2.5.4.23).telephoneNumber".

3.1.7. Referencing Contained Types

Sometimes the contents of a BIT STRING or OCTET STRING value are required to be the encodings of other ASN.1 values of specific ASN.1 types. For example, the extnValue component of the Extension type component in the Certificate type [11] is an OCTET STRING that is required to contain a Distinguished Encoding Rules (DER) [17] encoding of a certificate extension value. It is useful to be able to refer to the embedded encoded value and its components. An embedded encoded value is here referred to as a contained value and its associated type as the contained type.

If the ASN.1 type is a BIT STRING or OCTET STRING type containing encodings of other ASN.1 values then the <content> form of ComponentId may be used to identify the contained type. Subcomponents of that type can then be identified with further ComponentIds.

The contained type may be (effectively) an open type, constrained by some other component in an outer enclosing type (e.g., in a certificate Extension, extnValue is constrained by the chosen extnId). In these cases the next ComponentId, if any, MUST be of the <select> form.

For the purpose of building component references, the content of the extnValue OCTET STRING in the Extension type is assumed to be an open type having a notional component relation constraint with the extnId component as the single referenced component, i.e.,

```
EXTENSION.&ExtnType ({ExtensionSet}{@extnId})
```

The data-value component of the associated types for the EMBEDDED PDV and CHARACTER STRING types is an OCTET STRING containing the encoding of a data value described by the identification component. For the purpose of building component references, the content of the data-value OCTET STRING in these types is assumed to be an open type having a notional component relation constraint with the identification component as the single referenced component.

3.1.7.1. Contained Type Referencing Example

The Extension ASN.1 type [11] describes a single certificate extension value of a nominated extension type.

```
Extension ::= SEQUENCE {
    extnId     EXTENSION.&id ({ExtensionSet}),
    critical   BOOLEAN DEFAULT FALSE,
    extnValue  OCTET STRING
        -- contains a DER encoding of a value of type &ExtnType
        -- for the extension object identified by extnId -- }
```

EXTENSION.&id denotes an OBJECT IDENTIFIER and ({ExtensionSet}) constrains the OBJECT IDENTIFIER to be the identifier of a supported certificate extension.

The component reference "extnValue" on Extension refers to a component type of OCTET STRING. The corresponding component values will be OCTET STRING values. The component reference "extnValue.content" on Extension refers to the type of the contained type, which in this case is an open type.

One of the X.509 [11] standard extensions is basicConstraints, which is identified with the OBJECT IDENTIFIER 2.5.29.19 and is defined to have the following syntax.

```
BasicConstraintsSyntax ::= SEQUENCE {
    cA                  BOOLEAN DEFAULT FALSE,
    pathLenConstraint   INTEGER (0..MAX) OPTIONAL }
```

The component reference "extnValue.content.(2.5.29.19)" on Extension specifies a BasicConstraintsSyntax extension value and the component reference "extnValue.content.(2.5.29.19).cA" identifies the cA component of a BasicConstraintsSyntax extension value.

3.2. Matching of Components

The rule in a ComponentAssertion specifies how the zero, one or more component values identified by the component reference are tested by the assertion. Attribute matching rules are used to specify the semantics of the test.

Each matching rule has a notional set of attribute syntaxes (typically one), defined as ASN.1 types, to which it may be applied. When used in a ComponentAssertion these matching rules apply to the same ASN.1 types, only in this context the corresponding ASN.1 values are not necessarily complete attribute values.

Note that the referenced component type may be a tagged and/or constrained version of the expected attribute syntax (e.g., [0] INTEGER, whereas integerMatch would expect simply INTEGER), or an open type. Additional type substitutions of the kind described in

Section 3.1.1 are performed as required to reduce the component type to the same type as the attribute syntax expected by the matching rule.

If a matching rule applies to more than one attribute syntax (e.g., objectIdentifierFirstComponentMatch [12]) then the minimum number of substitutions required to conform to any one of those syntaxes is performed. If a matching rule can apply to any attribute syntax (e.g., the allComponentsMatch rule defined in Section 6.2) then the referenced component type is used as is, with no additional substitutions.

The value in a ComponentAssertion will be of the assertion syntax (i.e., ASN.1 type) required by the chosen matching rule. Note that the assertion syntax of a matching rule is not necessarily the same as the attribute syntax(es) to which the rule may be applied.

Some matching rules do not have a fixed assertion syntax (e.g., allComponentsMatch). The required assertion syntax is determined in each instance of use by the syntax of the attribute type to which the matching rule is applied. For these rules the ASN.1 type of the referenced component is used in place of an attribute syntax to decide the required assertion syntax.

The ComponentAssertion is Undefined if:

a) the matching rule in the ComponentAssertion is not known to the evaluating procedure,

b) the matching rule is not applicable to the referenced component type, even with the additional type substitutions,

c) the value in the ComponentAssertion does not conform to the assertion syntax defined for the matching rule,

d) some part of the component reference identifies an open type in the tested value that cannot be decoded, or

e) the implementation does not support the particular combination of component reference and matching rule.

If the ComponentAssertion is not Undefined then the ComponentAssertion evaluates to TRUE if there is at least one component value for which the matching rule applied to that component value returns TRUE, and evaluates to FALSE otherwise (which includes the case where there are no component values).

3.2.1. Applicability of Existing Matching Rules

3.2.1.1. String Matching

ASN.1 has a number of built in restricted character string types with different character sets and/or different character encodings. A directory user generally has little interest in the particular character set or encoding used to represent a character string component value, and some directory server implementations make no distinction between the different string types in their internal representation of values. So rather than define string matching rules for each of the restricted character string types, the existing case ignore and case exact string matching rules are extended to apply to component values of any of the restricted character string types and any ChoiceOfStrings type [9], in addition to component values of the DirectoryString type. This extension is only for the purposes of component matching described in this document.

The relevant string matching rules are: caseIgnoreMatch, caseIgnoreOrderingMatch, caseIgnoreSubstringsMatch, caseExactMatch, caseExactOrderingMatch and caseExactSubstringsMatch. The relevant restricted character string types are: NumericString, PrintableString, VisibleString, IA5String, UTF8String, BMPString, UniversalString, TeletexString, VideotexString, GraphicString and GeneralString. A ChoiceOfStrings type is a purely syntactic CHOICE of these ASN.1 string types. Note that GSER [9] declares each and every use of the DirectoryString{} parameterized type to be a ChoiceOfStrings type.

The assertion syntax of the string matching rules is still DirectoryString regardless of the string syntax of the component being matched. Thus an implementation will be called upon to compare a DirectoryString value to a value of one of the restricted character string types, or a ChoiceOfStrings type. As is the case when comparing two DirectoryStrings where the chosen alternatives are of different string types, the comparison proceeds so long as the corresponding characters are representable in both character sets. Otherwise matching returns FALSE.

3.2.1.2. Telephone Number Matching

Early editions of X.520 [12] gave the syntax of the telephoneNumber attribute as a constrained PrintableString. The fourth edition of X.520 equates the ASN.1 type name TelephoneNumber to the constrained PrintableString and uses TelephoneNumber as the attribute and assertion syntax. For the purposes of component matching,

telephoneNumberMatch and telephoneNumberSubstringsMatch are permitted to be applied to any PrintableString value, as well as to TelephoneNumber values.

3.2.1.3. Distinguished Name Matching

The DistinguishedName type is defined by assignment to be the same as the RDNSequence type, however RDNSequence is sometimes directly used in other type definitions. For the purposes of component matching, distinguishedNameMatch is also permitted to be applied to values of the RDNSequence type.

3.2.2. Additional Useful Matching Rules

This section defines additional matching rules that may prove useful in ComponentAssertions. These rules may also be used in extensibleMatch search filters [3].

3.2.2.1. The rdnMatch Matching Rule

The distinguishedNameMatch matching rule can match whole distinguished names but it is sometimes useful to be able to match specific Relative Distinguished Names (RDNs) in a Distinguished Name (DN) without regard for the other RDNs in the DN. The rdnMatch matching rule allows component RDNs of a DN to be tested.

The LDAP-style definitions for rdnMatch and its assertion syntax are:

```
( 1.2.36.79672281.1.13.3 NAME 'rdnMatch'
    SYNTAX 1.2.36.79672281.1.5.0 )

( 1.2.36.79672281.1.5.0 DESC 'RDN' )
```

The LDAP-specific encoding for a value of the RDN syntax is given by the <RelativeDistinguishedNameValue> rule [9].

The X.500-style definition for rdnMatch is:

```
rdnMatch MATCHING-RULE ::= {
    SYNTAX  RelativeDistinguishedName
    ID      { 1 2 36 79672281 1 13 3 } }
```

The rdnMatch rule evaluates to true if the component value and assertion value are the same RDN, using the same RDN comparison method as distinguishedNameMatch.

When using rdnMatch to match components of DNs it is important to note that the LDAP-specific encoding of a DN [5] reverses the order of the RDNs. So for the DN represented in LDAP as "cn=Steven Legg,o=Adacel,c=AU", the RDN "cn=Steven Legg" corresponds to the component reference "3", or alternatively, "-1".

3.2.2.2. The presentMatch Matching Rule

At times it would be useful to test not if a specific value of a particular component is present, but whether any value of a particular component is present. The presentMatch matching rule allows the presence of a particular component value to be tested.

The LDAP-style definitions for presentMatch and its assertion syntax are:

```
( 1.2.36.79672281.1.13.5 NAME 'presentMatch'
    SYNTAX 1.2.36.79672281.1.5.1 )

( 1.2.36.79672281.1.5.1 DESC 'NULL' )
```

The LDAP-specific encoding for a value of the NULL syntax is given by the <NullValue> rule [9].

The X.500-style definition for presentMatch is:

```
presentMatch MATCHING-RULE ::= {
    SYNTAX  NULL
    ID      { 1 2 36 79672281 1 13 5 } }
```

When used in a extensible match filter item, presentMatch behaves like the "present" case of a regular search filter. In a ComponentAssertion, presentMatch evaluates to TRUE if and only if the component reference identifies one or more component values, regardless of the actual component value contents. Note that if useDefaultValues is TRUE then the identified component values may be (part of) a DEFAULT value.

The notional count referenced by the <count> form of ComponentId is taken to be present if the SET OF value is present, and absent otherwise. Note that in ASN.1 notation an absent SET OF value is distinctly different from a SET OF value that is present but empty. It is up to the specification using the ASN.1 notation to decide whether the distinction matters. Often an empty SET OF component and an absent SET OF component are treated as semantically equivalent. If a SET OF value is present, but empty, a presentMatch on the SET OF component SHALL return TRUE and the notional count SHALL be regarded as present and equal to zero.

3.2.3. Summary of Useful Matching Rules

The following is a non-exhaustive list of useful matching rules and the ASN.1 types to which they can be applied, taking account of all the extensions described in Section 3.2.1, and the new matching rules defined in Section 3.2.2.

Matching Rule	ASN.1 Type
bitStringMatch	BIT STRING
booleanMatch	BOOLEAN
caseIgnoreMatch caseIgnoreOrderingMatch caseIgnoreSubstringsMatch caseExactMatch caseExactOrderingMatch caseExactSubstringsMatch	NumericString PrintableString VisibleString (ISO646String) IA5String UTF8String BMPString (UCS-2, UNICODE) UniversalString (UCS-4) TeletexString (T61String) VideotexString GraphicString GeneralString any ChoiceOfStrings type
caseIgnoreIA5Match caseExactIA5Match	IA5String
distinguishedNameMatch	DistinguishedName RDNSequence
generalizedTimeMatch generalizedTimeOrderingMatch	GeneralizedTime
integerMatch integerOrderingMatch	INTEGER
numericStringMatch numericStringOrderingMatch numericStringSubstringsMatch	NumericString
objectIdentifierMatch	OBJECT IDENTIFIER
octetStringMatch octetStringOrderingMatch octetStringSubstringsMatch	OCTET STRING

presentMatch	any ASN.1 type
rdnMatch	RelativeDistinguishedName
telephoneNumberMatch telephoneNumberSubstringsMatch	PrintableString TelephoneNumber
uTCTimeMatch uTCTimeOrderingMatch	UTCTime

Note that the allComponentsMatch matching rule defined in Section 6.2 can be used for equality matching of values of the ENUMERATED, NULL, REAL and RELATIVE-OID ASN.1 types, among other things.

4. ComponentFilter

The ComponentAssertion allows the value(s) of any one component type in a complex ASN.1 type to be matched, but there is often a desire to match the values of more than one component type. A ComponentFilter is an assertion about the presence, or values of, multiple components within an ASN.1 value.

The ComponentFilter assertion, an expression of ComponentAssertions, evaluates to either TRUE, FALSE or Undefined for each tested ASN.1 value.

A ComponentFilter is described by the following ASN.1 type (assumed to be defined with "EXPLICIT TAGS" in force):

```
ComponentFilter ::= CHOICE {
    item  [0] ComponentAssertion,
    and   [1] SEQUENCE OF ComponentFilter,
    or    [2] SEQUENCE OF ComponentFilter,
    not   [3] ComponentFilter }
```

Note: despite the use of SEQUENCE OF instead of SET OF for the "and" and "or" alternatives in ComponentFilter, the order of the component filters is not significant.

A ComponentFilter that is a ComponentAssertion evaluates to TRUE if the ComponentAssertion is TRUE, evaluates to FALSE if the ComponentAssertion is FALSE, and evaluates to Undefined otherwise.

The "and" of a sequence of component filters evaluates to TRUE if the sequence is empty or if each component filter evaluates to TRUE, evaluates to FALSE if at least one component filter is FALSE, and evaluates to Undefined otherwise.

The "or" of a sequence of component filters evaluates to FALSE if the sequence is empty or if each component filter evaluates to FALSE, evaluates to TRUE if at least one component filter is TRUE, and evaluates to Undefined otherwise.

The "not" of a component filter evaluates to TRUE if the component filter is FALSE, evaluates to FALSE if the component filter is TRUE, and evaluates to Undefined otherwise.

5. The componentFilterMatch Matching Rule

The componentFilterMatch matching rule allows a ComponentFilter to be applied to an attribute value. The result of the matching rule is the result of applying the ComponentFilter to the attribute value.

The LDAP-style definitions for componentFilterMatch and its assertion syntax are:

```
( 1.2.36.79672281.1.13.2 NAME 'componentFilterMatch'
    SYNTAX 1.2.36.79672281.1.5.2 )

( 1.2.36.79672281.1.5.2 DESC 'ComponentFilter' )
```

The LDAP-specific encoding for the ComponentFilter assertion syntax is specified by GSER [9].

As a convenience to implementors, an equivalent ABNF description of the GSER encoding for ComponentFilter is provided here. In the event that there is a discrepancy between this ABNF and the encoding determined by GSER, GSER is to be taken as definitive. The GSER encoding of a ComponentFilter is described by the following equivalent ABNF:

```
ComponentFilter = filter-item /
                  and-filter /
                  or-filter /
                  not-filter

filter-item     = item-chosen ComponentAssertion
and-filter      = and-chosen  SequenceOfComponentFilter
or-filter       = or-chosen   SequenceOfComponentFilter
not-filter      = not-chosen  ComponentFilter
```

```
   item-chosen     = %x69.74.65.6D.3A  ; "item:"
   and-chosen      = %x61.6E.64.3A     ; "and:"
   or-chosen       = %x6F.72.3A        ; "or:"
   not-chosen      = %x6E.6F.74.3A     ; "not:"

   SequenceOfComponentFilter = "{" [ sp ComponentFilter
                                  *( "," sp ComponentFilter) ] sp "}"

   ComponentAssertion = "{" [ sp component "," ]
                            [ sp useDefaultValues "," ]
                              sp rule ","
                              sp assertion-value sp "}"
   component          = component-label msp StringValue
   useDefaultValues   = use-defaults-label msp BooleanValue
   rule               = rule-label msp ObjectIdentifierValue
   assertion-value    = value-label msp Value

   component-label    = %x63.6F.6D.70.6F.6E.65.6E.74  ; "component"
   use-defaults-label = %x75.73.65.44.65.66.61.75.6C.74.56.61.6C.75
                        %x65.73                    ; "useDefaultValues"
   rule-label         = %x72.75.6C.65             ; "rule"
   value-label        = %x76.61.6C.75.65          ; "value"

   sp                 =  *%x20  ; zero, one or more space characters
   msp                = 1*%x20  ; one or more space characters
```

The ABNF for <Value>, <StringValue>, <ObjectIdentifierValue> and <BooleanValue> is defined by GSER [9].

The ABNF descriptions of LDAP-specific encodings for attribute syntaxes typically do not clearly or consistently delineate the component parts of an attribute value. A regular and uniform character string encoding for arbitrary component data types is needed to encode the assertion value in a ComponentAssertion. The <Value> rule from GSER provides a human readable text encoding for a component value of any arbitrary ASN.1 type.

The X.500-style definition [10] for componentFilterMatch is:

```
   componentFilterMatch MATCHING-RULE ::= {
       SYNTAX  ComponentFilter
       ID      { 1 2 36 79672281 1 13 2 } }
```

A ComponentAssertion can potentially use any matching rule, including componentFilterMatch, so componentFilterMatch may be nested. The component references in a nested componentFilterMatch are relative to

the component corresponding to the containing ComponentAssertion. In Section 7, an example search on the seeAlso attribute shows this usage.

6. Equality Matching of Complex Components

It is possible to test if an attribute value of a complex ASN.1 syntax is the same as some purported (i.e., assertion) value by using a complicated ComponentFilter that tests if corresponding components are the same. However, it would be more convenient to be able to present a whole assertion value to a matching rule that could do the component-wise comparison of an attribute value with the assertion value for any arbitrary attribute syntax. Similarly, the ability to do a straightforward equality comparison of a component value that is itself of a complex ASN.1 type would also be convenient.

It would be difficult to define a single matching rule that simultaneously satisfies all notions of what the equality matching semantics should be. For example, in some instances a case sensitive comparison of string components may be preferable to a case insensitive comparison. Therefore a basic equality matching rule, allComponentsMatch, is defined in Section 6.2, and the means to derive new matching rules from it with slightly different equality matching semantics are described in Section 6.3.

The directoryComponentsMatch defined in Section 6.4 is a derivation of allComponentsMatch that suits typical uses of the directory. Other specifications are free to derive new rules from allComponentsMatch or directoryComponentsMatch, that suit their usage of the directory.

The allComponentsMatch rule, the directoryComponentsMatch rule and any matching rules derived from them are collectively called component equality matching rules.

6.1. The OpenAssertionType Syntax

The component equality matching rules have a variable assertion syntax. In X.500 this is indicated by omitting the optional SYNTAX field in the MATCHING-RULE information object. The assertion syntax then defaults to the target attribute's syntax in actual usage, unless the description of the matching rule says otherwise. The SYNTAX field in the LDAP-specific encoding of a MatchingRuleDescription is mandatory, so the OpenAssertionType syntax is defined to fill the same role. That is, the OpenAssertionType syntax is semantically equivalent to an omitted SYNTAX field in an X.500 MATCHING-RULE information object. OpenAssertionType MUST NOT be used as the attribute syntax in an attribute type definition.

Unless explicitly varied by the description of a particular matching rule, if an OpenAssertionType assertion value appears in a ComponentAssertion its LDAP-specific encoding is described by the <Value> rule in GSER [9], otherwise its LDAP-specific encoding is the encoding defined for the syntax of the attribute type to which the matching rule with the OpenAssertionType assertion syntax is applied.

The LDAP definition for the OpenAssertionType syntax is:

```
( 1.2.36.79672281.1.5.3 DESC 'OpenAssertionType' )
```

6.2. The allComponentsMatch Matching Rule

The LDAP-style definition for allComponentsMatch is:

```
( 1.2.36.79672281.1.13.6 NAME 'allComponentsMatch'
    SYNTAX 1.2.36.79672281.1.5.3 )
```

The X.500-style definition for allComponentsMatch is:

```
allComponentsMatch MATCHING-RULE ::= {
    ID      { 1 2 36 79672281 1 13 6 } }
```

When allComponentsMatch is used in a ComponentAssertion the assertion syntax is the same as the ASN.1 type of the identified component. Otherwise, the assertion syntax of allComponentsMatch is the same as the attribute syntax of the attribute to which the matching rule is applied.

Broadly speaking, this matching rule evaluates to true if and only if corresponding components of the assertion value and the attribute or component value are the same.

In detail, equality is determined by the following cases applied recursively.

a) Two values of a SET or SEQUENCE type are the same if and only if, for each component type, the corresponding component values are either,

   1) both absent,

   2) both present and the same, or

   3) absent or the same as the DEFAULT value for the component, if a DEFAULT value is defined.

Values of an EMBEDDED PDV, EXTERNAL, unrestricted CHARACTER STRING, or INSTANCE OF type are compared according to their respective associated SEQUENCE type (see Section 3.1.2).

b) Two values of a SEQUENCE OF type are the same if and only if, the values have the same number of (possibly duplicated) instances and corresponding instances are the same.

c) Two values of a SET OF type are the same if and only if, the values have the same number of instances and each distinct instance occurs in both values the same number of times, i.e., both values have the same instances, including duplicates, but in any order.

d) Two values of a CHOICE type are the same if and only if, both values are of the same chosen alternative and the component values are the same.

e) Two BIT STRING values are the same if and only if the values have the same number of bits and corresponding bits are the same. If the BIT STRING type is defined with a named bit list then trailing zero bits in the values are treated as absent for the purposes of this comparison.

f) Two BOOLEAN values are the same if and only if both are TRUE or both are FALSE.

g) Two values of a string type are the same if and only if the values have the same number of characters and corresponding characters are the same. Letter case is significant. For the purposes of allComponentsMatch, the string types are NumericString, PrintableString, TeletexString (T61String), VideotexString, IA5String, GraphicString, VisibleString (ISO646String), GeneralString, UniversalString, BMPString, UTF8String, GeneralizedTime, UTCTime and ObjectDescriptor.

h) Two INTEGER values are the same if and only if the integers are equal.

i) Two ENUMERATED values are the same if and only if the enumeration item identifiers are the same (equivalently, if the integer values associated with the identifiers are equal).

j) Two NULL values are always the same, unconditionally.

k) Two OBJECT IDENTIFIER values are the same if and only if the values have the same number of arcs and corresponding arcs are the same.

l) Two OCTET STRING values are the same if and only if the values have the same number of octets and corresponding octets are the same.

m) Two REAL values are the same if and only if they are both the same special value, or neither is a special value and they have the same base and represent the same real number. The special values for REAL are zero, PLUS-INFINITY and MINUS-INFINITY.

n) Two RELATIVE-OID values are the same if and only if the values have the same number of arcs and corresponding arcs are the same. The respective starting nodes for the RELATIVE-OID values are disregarded in the comparison, i.e., they are assumed to be the same.

o) Two values of an open type are the same if and only if both are of the same ASN.1 type and are the same according to that type. If the actual ASN.1 type of the values is unknown then the allComponentsMatch rule evaluates to Undefined.

Tags and constraints, being part of the type definition and not part of the abstract values, are ignored for matching purposes.

The allComponentsMatch rule may be used as the defined equality matching rule for an attribute.

6.3. Deriving Component Equality Matching Rules

A new component equality matching rule with more refined matching semantics may be derived from allComponentsMatch, or any other component equality matching rule, using the convention described in this section.

The matching behaviour of a derived component equality matching rule is specified by nominating, for each of one or more identified components, a commutative equality matching rule that will be used to match values of that component. This overrides the matching that would otherwise occur for values of that component using the base rule for the derivation. These overrides can be conveniently represented as rows in a table of the following form.

Component	Matching Rule

Usually, all component values of a particular ASN.1 type are to be matched the same way. An ASN.1 type reference (e.g., DistinguishedName) or an ASN.1 built-in type name (e.g., INTEGER) in the Component column of the table specifies that the nominated equality matching rule is to be applied to all values of the named type, regardless of context.

An ASN.1 type reference with a component reference appended (separated by a ".") specifies that the nominated matching rule applies only to the identified components of values of the named type. Other component values that happen to be of the same ASN.1 type are not selected.

Additional type substitutions as described in Section 3.2 are assumed to be performed to align the component type with the matching rule assertion syntax.

Conceptually, the rows in a table for the base rule are appended to the rows in the table for a derived rule for the purpose of deciding the matching semantics of the derived rule. Notionally, allComponentsMatch has an empty table.

A row specifying values of an outer containing type (e.g., DistinguishedName) takes precedence over a row specifying values of an inner component type (e.g., RelativeDistinguishedName), regardless of their order in the table. Specifying a row for component values of an inner type is only useful if a value of the type can also appear on its own, or as a component of values of a different outer type. For example, if there is a row for DistinguishedName then a row for RelativeDistinguishedName can only ever apply to RelativeDistinguishedName component values that are not part of a DistinguishedName. A row for values of an outer type in the table for the base rule takes precedence over a row for values of an inner type in the table for the derived rule.

Where more than one row applies to a particular component value the earlier row takes precedence over the later row. Thus rows in the table for the derived rule take precedence over any rows for the same component in the table for the base rule.

6.4. The directoryComponentsMatch Matching Rule

The directoryComponentsMatch matching rule is derived from the allComponentsMatch matching rule.

The LDAP-style definition for directoryComponentsMatch is:

```
( 1.2.36.79672281.1.13.7 NAME 'directoryComponentsMatch'
    SYNTAX 1.2.36.79672281.1.5.3 )
```

The X.500-style definition for directoryComponentsMatch is:

```
directoryComponentsMatch MATCHING-RULE ::= {
    ID      { 1 2 36 79672281 1 13 7 } }
```

The matching semantics of directoryComponentsMatch are described by the following table, using the convention described in Section 6.3.

ASN.1 Type	Matching Rule
RDNSequence	distinguishedNameMatch
RelativeDistinguishedName	rdnMatch
TelephoneNumber	telephoneNumberMatch
FacsimileTelephoneNumber.telephoneNumber	telephoneNumberMatch
NumericString	numericStringMatch
GeneralizedTime	generalizedTimeMatch
UTCTime	uTCTimeMatch
DirectoryString{}	caseIgnoreMatch
BMPString	caseIgnoreMatch
GeneralString	caseIgnoreMatch
GraphicString	caseIgnoreMatch
IA5String	caseIgnoreMatch
PrintableString	caseIgnoreMatch
TeletexString	caseIgnoreMatch
UniversalString	caseIgnoreMatch
UTF8String	caseIgnoreMatch
VideotexString	caseIgnoreMatch
VisibleString	caseIgnoreMatch

Notes:

1) The DistinguishedName type is defined by assignment to be the same as the RDNSequence type. Some types (e.g., Name and LocalName) directly reference RDNSequence rather than DistinguishedName. Specifying RDNSequence captures all these DN-like types.

2) A RelativeDistinguishedName value is only matched by rdnMatch if it is not part of an RDNSequence value.

3) The telephone number component of the FacsimileTelephoneNumber ASN.1 type [12] is defined as a constrained PrintableString. PrintableString component values that are part of a FacsimileTelephoneNumber value can be identified separately from

other components of PrintableString type by the specifier FacsimileTelephoneNumber.telephoneNumber, so that telephoneNumberMatch can be selectively applied. The fourth edition of X.520 defines the telephoneNumber component of FacsimileTelephoneNumber to be of the type TelephoneNumber, making the row for FacsimileTelephoneNumber.telephoneNumber components redundant.

The directoryComponentsMatch rule may be used as the defined equality matching rule for an attribute.

7. Component Matching Examples

This section contains examples of search filters using the componentFilterMatch matching rule. The filters are described using the string representation of LDAP search filters [18]. Note that this representation requires asterisks to be escaped in assertion values (in these examples the assertion values are all <ComponentAssertion> encodings). The asterisks have not been escaped in these examples for the sake of clarity, and to avoid confusing the protocol representation of LDAP search filter assertion values, where such escaping does not apply. Line breaks and indenting have been added only as an aid to readability.

The example search filters using componentFilterMatch are all single extensible match filter items, though there is no reason why componentFilterMatch can't be used in more complicated search filters.

The first examples describe searches over the objectClasses schema operational attribute, which has an attribute syntax described by the ASN.1 type ObjectClassDescription [10], and holds the definitions of the object classes known to a directory server. The definition of ObjectClassDescription is as follows:

```
ObjectClassDescription ::= SEQUENCE {
    identifier       OBJECT-CLASS.&id,
    name             SET OF DirectoryString {ub-schema} OPTIONAL,
    description      DirectoryString {ub-schema} OPTIONAL,
    obsolete         BOOLEAN DEFAULT FALSE,
    information  [0] ObjectClassInformation }

ObjectClassInformation ::= SEQUENCE {
    subclassOf       SET OF OBJECT-CLASS.&id OPTIONAL,
    kind             ObjectClassKind DEFAULT structural,
    mandatories  [3] SET OF ATTRIBUTE.&id OPTIONAL,
    optionals    [4] SET OF ATTRIBUTE.&id OPTIONAL }
```

```
   ObjectClassKind ::= ENUMERATED {
       abstract     (0),
       structural   (1),
       auxiliary    (2) }
```

OBJECT-CLASS.&id and ATTRIBUTE.&id are equivalent to the OBJECT IDENTIFIER ASN.1 type. A value of OBJECT-CLASS.&id is an OBJECT IDENTIFIER for an object class. A value of ATTRIBUTE.&id is an OBJECT IDENTIFIER for an attribute type.

The following search filter finds the object class definition for the object class identified by the OBJECT IDENTIFIER 2.5.6.18:

```
   (objectClasses:componentFilterMatch:=
        item:{ component "identifier",
               rule objectIdentifierMatch, value 2.5.6.18 })
```

A match on the "identifier" component of objectClasses values is equivalent to the objectIdentifierFirstComponentMatch matching rule applied to attribute values of the objectClasses attribute type. The componentFilterMatch matching rule subsumes the functionality of the objectIdentifierFirstComponentMatch, integerFirstComponentMatch and directoryStringFirstComponentMatch matching rules.

The following search filter finds the object class definition for the object class called foobar:

```
   (objectClasses:componentFilterMatch:=
       item:{ component "name.*",
              rule caseIgnoreMatch, value "foobar" })
```

An object class definition can have multiple names and the above filter will match an objectClasses value if any one of the names is "foobar".

The component reference "name.0" identifies the notional count of the number of names in an object class definition. The following search filter finds object class definitions with exactly one name:

```
   (objectClasses:componentFilterMatch:=
       item:{ component "name.0", rule integerMatch, value 1 })
```

The "description" component of an ObjectClassDescription is defined to be an OPTIONAL DirectoryString. The following search filter finds object class definitions that have descriptions, regardless of the contents of the description string:

```
   (objectClasses:componentFilterMatch:=
       item:{ component "description",
              rule presentMatch, value NULL })
```

The presentMatch returns TRUE if the description component is present and FALSE otherwise.

The following search filter finds object class definitions that don't have descriptions:

```
   (objectClasses:componentFilterMatch:=
       not:item:{ component "description",
                  rule presentMatch, value NULL })
```

The following search filter finds object class definitions with the word "bogus" in the description:

```
   (objectClasses:componentFilterMatch:=
       item:{ component "description",
              rule caseIgnoreSubstringsMatch,
              value { any:"bogus" } })
```

The assertion value is of the SubstringAssertion syntax, i.e.,

```
   SubstringAssertion ::= SEQUENCE OF CHOICE {
       initial      [0] DirectoryString {ub-match},
       any          [1] DirectoryString {ub-match},
       final        [2] DirectoryString {ub-match} }
```

The "obsolete" component of an ObjectClassDescription is defined to be DEFAULT FALSE. An object class is obsolete if the "obsolete" component is present and set to TRUE. The following search filter finds all obsolete object classes:

```
   (objectClasses:componentFilterMatch:=
       item:{ component "obsolete", rule booleanMatch, value TRUE })
```

An object class is not obsolete if the "obsolete" component is not present, in which case it defaults to FALSE, or is present but is explicitly set to FALSE. The following search filter finds all non-obsolete object classes:

```
   (objectClasses:componentFilterMatch:=
       item:{ component "obsolete", rule booleanMatch, value FALSE })
```

The useDefaultValues flag in the ComponentAssertion defaults to TRUE so the componentFilterMatch rule treats an absent "obsolete" component as being present and set to FALSE. The following search

filter finds only object class definitions where the "obsolete" component has been explicitly set to FALSE, rather than implicitly defaulting to FALSE:

```
(objectClasses:componentFilterMatch:=
    item:{ component "obsolete", useDefaultValues FALSE,
           rule booleanMatch, value FALSE })
```

With the useDefaultValues flag set to FALSE, if the "obsolete" component is absent the component reference identifies no component value and the matching rule will return FALSE. The matching rule can only return TRUE if the component is present and set to FALSE.

The "information.kind" component of the ObjectClassDescription is an ENUMERATED type. The allComponentsMatch matching rule can be used to match values of an ENUMERATED type. The following search filter finds object class definitions for auxiliary object classes:

```
(objectClasses:componentFilterMatch:=
    item:{ component "information.kind",
           rule allComponentsMatch, value auxiliary })
```

The following search filter finds auxiliary object classes with commonName (cn or 2.5.4.3) as a mandatory attribute:

```
(objectClasses:componentFilterMatch:=and:{
    item:{ component "information.kind",
           rule allComponentsMatch, value auxiliary },
    item:{ component "information.mandatories.*",
           rule objectIdentifierMatch, value cn } })
```

The following search filter finds auxiliary object classes with commonName as a mandatory or optional attribute:

```
(objectClasses:componentFilterMatch:=and:{
    item:{ component "information.kind",
           rule allComponentsMatch, value auxiliary },
    or:{
        item:{ component "information.mandatories.*",
               rule objectIdentifierMatch, value cn },
        item:{ component "information.optionals.*",
               rule objectIdentifierMatch, value cn } } })
```

Extra care is required when matching optional SEQUENCE OF or SET OF components because of the distinction between an absent list of instances and a present, but empty, list of instances. The following search filter finds object class definitions with less than three

names, including object class definitions with a present but empty list of names, but does not find object class definitions with an absent list of names:

```
(objectClasses:componentFilterMatch:=
    item:{ component "name.0",
           rule integerOrderingMatch, value 3 })
```

If the "name" component is absent the "name.0" component is also considered to be absent and the ComponentAssertion evaluates to FALSE. If the "name" component is present, but empty, the "name.0" component is also present and equal to zero, so the ComponentAssertion evaluates to TRUE. To also find the object class definitions with an absent list of names the following search filter would be used:

```
(objectClasses:componentFilterMatch:=or:{
    not:item:{ component "name", rule presentMatch, value NULL },
    item:{ component "name.0",
           rule integerOrderingMatch, value 3 } })
```

Distinguished names embedded in other syntaxes can be matched with a componentFilterMatch. The uniqueMember attribute type has an attribute syntax described by the ASN.1 type NameAndOptionalUID.

```
NameAndOptionalUID ::= SEQUENCE {
    dn          DistinguishedName,
    uid         UniqueIdentifier OPTIONAL }
```

The following search filter finds values of the uniqueMember attribute containing the author's DN:

```
(uniqueMember:componentFilterMatch:=
    item:{ component "dn",
           rule distinguishedNameMatch,
           value "cn=Steven Legg,o=Adacel,c=AU" })
```

The DistinguishedName and RelativeDistinguishedName ASN.1 types are also complex ASN.1 types so the component matching rules can be applied to their inner components.

```
DistinguishedName   ::= RDNSequence

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET SIZE (1..MAX) OF
    AttributeTypeAndValue
```

```
   AttributeTypeAndValue ::= SEQUENCE {
       type        AttributeType ({SupportedAttributes}),
       value       AttributeValue ({SupportedAttributes}{@type}) }

   AttributeType ::= ATTRIBUTE.&id

   AttributeValue ::= ATTRIBUTE.&Type
```

ATTRIBUTE.&Type is an open type. A value of ATTRIBUTE.&Type is constrained by the type component of AttributeTypeAndValue to be of the attribute syntax of the nominated attribute type. Note: the fourth edition of X.500 extends and renames the AttributeTypeAndValue SEQUENCE type.

The seeAlso attribute has the DistinguishedName syntax. The following search filter finds seeAlso attribute values containing the RDN, "o=Adacel", anywhere in the DN:

```
   (seeAlso:componentFilterMatch:=
       item:{ component "*", rule rdnMatch, value "o=Adacel" })
```

The following search filter finds all seeAlso attribute values with "cn=Steven Legg" as the RDN of the named entry (i.e., the "first" RDN in an LDAPDN or the "last" RDN in an X.500 DN):

```
   (seeAlso:componentFilterMatch:=
       item:{ component "-1",
              rule rdnMatch, value "cn=Steven Legg" })
```

The following search filter finds all seeAlso attribute values naming entries in the DIT subtree of "o=Adacel,c=AU":

```
   (seeAlso:componentFilterMatch:=and:{
       item:{ component "1", rule rdnMatch, value "c=AU" },
       item:{ component "2", rule rdnMatch, value "o=Adacel" } })
```

The following search filter finds all seeAlso attribute values containing the naming attribute types commonName (cn) and telephoneNumber in the same RDN:

```
   (seeAlso:componentFilterMatch:=
       item:{ component "*", rule componentFilterMatch,
              value and:{
                  item:{ component "*.type",
                         rule objectIdentifierMatch, value cn },
                  item:{ component "*.type",
                         rule objectIdentifierMatch,
                         value telephoneNumber } } })
```

The following search filter would find all seeAlso attribute values containing the attribute types commonName and telephoneNumber, but not necessarily in the same RDN:

```
(seeAlso:componentFilterMatch:=and:{
    item:{ component "*.*.type",
           rule objectIdentifierMatch, value cn },
    item:{ component "*.*.type",
           rule objectIdentifierMatch, value telephoneNumber } })
```

The following search filter finds all seeAlso attribute values containing the word "Adacel" in any organizationalUnitName (ou) attribute value in any AttributeTypeAndValue of any RDN:

```
(seeAlso:componentFilterMatch:=
    item:{ component "*.*.value.(2.5.4.11)",
           rule caseIgnoreSubstringsMatch,
           value { any:"Adacel" } })
```

The component reference "*.*.value" identifies an open type, in this case an attribute value. In a particular AttributeTypeAndValue, if the attribute type is not organizationalUnitName then the ComponentAssertion evaluates to FALSE. Otherwise the substring assertion is evaluated against the attribute value.

Absent component references in ComponentAssertions can be exploited to avoid false positive matches on multi-valued attributes. For example, suppose there is a multi-valued attribute named productCodes, defined to have the Integer syntax (1.3.6.1.4.1.1466.115.121.1.27). Consider the following search filter:

```
(&(!(productCodes:integerOrderingMatch:=3))
  (productCodes:integerOrderingMatch:=8))
```

An entry whose productCodes attribute contains only the values 1 and 10 will match the above filter. The first subfilter is satisfied by the value 10 (10 is not less than 3), and the second subfilter is satisfied by the value 1 (1 is less than 8). The following search filter can be used instead to only match entries that have a productCodes value in the range 3 to 7, because the ComponentFilter is evaluated against each productCodes value in isolation:

```
(productCodes:componentFilterMatch:= and:{
     not:item:{ rule integerOrderingMatch, value 3 },
    item:{ rule integerOrderingMatch, value 8 } })
```

An entry whose productCodes attribute contains only the values 1 and 10 will not match the above filter.

8. Security Considerations

The component matching rules described in this document allow for a compact specification of matching capabilities that could otherwise have been defined by a plethora of specific matching rules, i.e., despite their expressiveness and flexibility the component matching rules do not behave in a way uncharacteristic of other matching rules, so the security issues for component matching rules are no different than for any other matching rule. However, because the component matching rules are applicable to any attribute syntax, support for them in a directory server may allow searching of attributes that were previously unsearchable by virtue of there not being a suitable matching rule. Such attribute types ought to be properly protected with appropriate access controls. A generic, interoperable access control mechanism has not yet been developed, however, and implementors should be aware of the interaction of that lack with the increased risk of exposure described above.

9. Acknowledgements

The author would like to thank Tom Gindin for private email discussions that clarified and refined the ideas presented in this document.

10. IANA Considerations

The Internet Assigned Numbers Authority (IANA) has updated the LDAP descriptors registry [8] as indicated by the following templates:

```
Subject: Request for LDAP Descriptor Registration
Descriptor (short name): componentFilterMatch
Object Identifier: 1.2.36.79672281.1.13.2
Person & email address to contact for further information:
  Steven Legg <steven.legg@adacel.com.au>
Usage: other (matching rule)
Specification: RFC 3687
Author/Change Controller: IESG
```

```
Subject: Request for LDAP Descriptor Registration
Descriptor (short name): rdnMatch
Object Identifier: 1.2.36.79672281.1.13.3
Person & email address to contact for further information:
  Steven Legg <steven.legg@adacel.com.au>
Usage: other (matching rule)
Specification: RFC 3687
Author/Change Controller: IESG

Subject: Request for LDAP Descriptor Registration
Descriptor (short name): presentMatch
Object Identifier: 1.2.36.79672281.1.13.5
Person & email address to contact for further information:
  Steven Legg <steven.legg@adacel.com.au>
Usage: other (matching rule)
Specification: RFC 3687
Author/Change Controller: IESG

Subject: Request for LDAP Descriptor Registration
Descriptor (short name): allComponentsMatch
Object Identifier: 1.2.36.79672281.1.13.6
Person & email address to contact for further information:
  Steven Legg <steven.legg@adacel.com.au>
Usage: other (matching rule)
Specification: RFC 3687
Author/Change Controller: IESG

Subject: Request for LDAP Descriptor Registration
Descriptor (short name): directoryComponentsMatch
Object Identifier: 1.2.36.79672281.1.13.7
Person & email address to contact for further information:
  Steven Legg <steven.legg@adacel.com.au>
Usage: other (matching rule)
Specification: RFC 3687
Author/Change Controller: IESG
```

The object identifiers have been assigned for use in this specification by Adacel Technologies, under an arc assigned to Adacel by Standards Australia.

11. References

11.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

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

[4] Wahl, M., Coulbeck, A., Howes, T. and S. Kille, "Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions", RFC 2252, December 1997.

[5] Wahl, M., Kille S. and T. Howes. "Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names", RFC 2253, December 1997.

[6] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[7] Hodges, J. and R. Morgan, "Lightweight Directory Access Protocol (v3): Technical Specification", RFC 3377, September 2002.

[8] Zeilenga, K., "Internet Assigned Numbers Authority (IANA) Considerations for the Lightweight Directory Access Protocol (LDAP)", BCP 64, RFC 3383, September 2002.

[9] Legg, S., "Generic String Encoding Rules (GSER) for ASN.1 Types", RFC 3641, October 2003.

[10] ITU-T Recommendation X.501 (1993) | ISO/IEC 9594-2:1994, Information Technology - Open Systems Interconnection - The Directory: Models

[11] ITU-T Recommendation X.509 (1997) | ISO/IEC 9594-8:1998, Information Technology - Open Systems Interconnection - The Directory: Authentication Framework

[12] ITU-T Recommendation X.520 (1993) | ISO/IEC 9594-6:1994, Information technology - Open Systems Interconnection - The Directory: Selected attribute types

[13] ITU-T Recommendation X.680 (07/02) | ISO/IEC 8824-1:2002, Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation

[14] ITU-T Recommendation X.681 (07/02) | ISO/IEC 8824-2:2002, Information technology - Abstract Syntax Notation One (ASN.1): Information object specification

[15] ITU-T Recommendation X.682 (07/02) | ISO/IEC 8824-3:2002, Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification

[16] ITU-T Recommendation X.683 (07/02) | ISO/IEC 8824-4:2002, Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications

[17] ITU-T Recommendation X.690 (07/02) | ISO/IEC 8825-1, Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)

12.2. Informative References

[18] Howes, T., "The String Representation of LDAP Search Filters", RFC 2254, December 1997.

[19] ITU-T Recommendation X.500 (1993) | ISO/IEC 9594-1:1994, Information Technology - Open Systems Interconnection - The Directory: Overview of concepts, models and services

12. Intellectual Property Statement

13. Author's Address

Steven Legg
Adacel Technologies Ltd.
250 Bay Street
Brighton, Victoria 3186
AUSTRALIA

Phone: +61 3 8530 7710
Fax: +61 3 8530 7888
EMail: steven.legg@adacel.com.au

14. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.