ConfigurationManagement:1

Service Template Version 1.01

For UPnP Version 1.0

Status: ***Standardized DCP (SDCP)***

Date: ***July 20, 2010***

This Standardized DCP has been adopted as a Standardized DCP by the Steering Committee of the UPnP Forum, pursuant to Section 2.1(c)(ii) of the UPnP Forum Membership Agreement. UPnP Forum Members have rights and licenses defined by Section 3 of the UPnP Forum Membership Agreement to use and reproduce the Standardized DCP in UPnP Compliant Devices. All such use is subject to all of the provisions of the UPnP Forum Membership Agreement.

Authors	Company
André Bottaro	France Telecom Group
Enrico Grosso	Telecom Italia
Levent Gurgen	France Telecom Group
William Lupton	2Wire
Davide Moreo (Editor)	Telecom Italia
François-Gaël Ottogalli	France Telecom Group
Kiran Vedula	Samsung Electronics

* Note: The UPnP Forum in no way guarantees the accuracy or completeness of this author list and in no way implies any rights for or support from those members listed. This list is not the specifications' contributor list that is kept on the UPnP Forum's website.

Contents

List of Tables

1. Overview and Scope

This service definition is compliant with the UPnP Device Architecture version *1.0*. It defines a service type referred to herein as *ConfigurationManagement:1* service.

1.1. Introduction

The *ConfigurationManagement:1* Service (CMS) defines a generic UPnP service, hosted by an UPnP *Parent Device*, that allows a control point to manage the configuration in terms of parameters supported by the device and their actual values.

The term *Parent Device* is frequently used thorough this document. It refers to UPnP device/service sub-tree whose root is the UPnP device that contains the *ConfigurationManagement:1* service instance. UPnP actions or other operations on a *Parent Device* SHOULD apply to all levels of this sub-tree, but SHOULD NOT apply to an embedded device that itself contains a *ConfigurationManagement:1* service instance.

Parameters may describe configuration features of the device or may be related to status information of the *Parent Device*. CMS defines the concept of *data model* as the set of parameters provided by a device for being managed by CMS actions.

CMS can be used as a UPnP service in any UPnP Device, whether the UPnP DM *ManageableDevice* or a UPnP Device defined by another UPnP Working Committee. Refer to [DEVICE] for details about the possible deployment scenarios.

This document specifies two related concepts:

- Generic actions for managing *Parent Device* configuration.

- A basic set of configuration parameters that a *Parent Device* may support. Such configuration parameters are referred as *Common Objects*; additional configuration parameters may be defined by other UPnP DCPs, by other organizations'data model definitions or by vendor specific extensions.

This service-type enables the following functions:

- Reading the actual configuration and status of a *Parent Device* using CMS (i.e. "reading" parameters), in terms of available *data model* parameters with their values.
- Changing the actual configuration of a *Parent Device* using CMS (i.e. "writing" parameters), by setting new values of parameters and creating or deleting object instances (i.e. rows in parameter tables).

The CMS is mandatory for all *ManageableDevices* and is required for every UPnP devices supporting a data model for configuration purposes.

This service template does **not** address:

- *Dynamic creation and deletion of sets of Parameters (i.e. only Parameters values may be changed using this service and creating or deleting object instances, for example adding or deleting table rows).*

1.2. References

This section lists the normative references used in the UPnP DM specifications and includes the tag inside square brackets that is used for each such reference:

[UDA] *UPnP Device Architecture, version1.0*, UPnP Forum, July 20, 2006. Available at: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf

[DEVICE] *UPnP ManageableDevice:1 Device Document*, UPnP Forum, July 20, 2010, http://www.upnp.org/specs/dm/UPnP-dm-ManageableDevice-v1.0-Device.pdf

[BMS] *UPnP BasicManagement:1 Service Document*, UPnP Forum, July 20, 2010, http://www.upnp.org/specs/dm/UPnP-dm-BasicManagement-v1.0-Service.pdf

[SMS] *UPnP SoftwareManagement:1 Service Document*, UPnP Forum, July 20, 2010, http://www.upnp.org/specs/dm/UPnP-dm-SoftwareManagement-v1.0-Service.pdf

[EBNF] *W3C Extensible Markup Language (XML) 1.0 (Fifth Edition) -Notation section, http://www.w3.org/TR/REC-xml#sec-notation*

[RFC 2119] RFC 2119, *Key words for use in RFCs to Indicate Requirement Levels*, March 1997, http://tools.ietf.org/html/rfc2119

[RFC 3513] RFC 3513, *Internet Protocol Version 6 (IPv6) Addressing Architecture*, IETF, April 2003, http://tools.ietf.org/html/rfc3513

[SOAP] *Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/2000/NOTE-SOAP-20000508*

[XML] *Extensible Markup Language (XML) 1.0 (Fourth Edition), http://www.w3.org/TR/REC-xml*

[XML-NCName] *W3C XML Schema Part 2: Datatypes Second Edition, http://www.w3.org/TR/xmlschema-2/#NCName. NCName syntax defined in: http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName*

[IANA-MIME] *MIME Media Types registered at IANA: http://www.iana.org/assignments/media-types/*

[URI] *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*, IETF, January 2005, http://tools.ietf.org/html/rfc3986

[XML-NS] *The "xml:" Namespace*, W3C, April 2006, http://www.w3.org/XML/1998/namespace

[XML-NMSP] *Namespaces in XML*, W3C, August 2006, http://www.w3.org/TR/REC-xml-names

[XML-SCHEMA-1] *XML Schema Part 1: Structures Second Edition*, W3C, October 2004, http://www.w3.org/TR/xmlschema-1

[XML-SCHEMA-2] *XML Schema Part 2: Datatypes Second Edition*, W3C, October 2004, http://www.w3.org/TR/xmlschema-2

1.3. Glossary

BMS *BasicManagement* Service

CMS *ConfigurationManagement* Service

SMS *SoftwareManagement* Service

CSV Comma Separated Value

BNF Backus-Naur Form

DM Device Management

MD *ManageableDevice*

DU Deployment Unit

XSD XML Schema Definition

1.4. Notation

In this document, features are described as Required, Recommended, or Optional as follows:

the keywords "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this specification are to be interpreted as described in [[RFC 2119]].

In addition, the following keywords are used in this specification:

PROHIBITED – The definition or behavior is an absolute prohibition of this specification. Opposite of REQUIRED.

CONDITIONALLY REQUIRED – The definition or behavior depends on a condition. If the specified condition is met, then the definition or behavior is REQUIRED, otherwise it is PROHIBITED.

CONDITIONALLY OPTIONAL – The definition or behavior depends on a condition. If the specified condition is met, then the definition or behavior is OPTIONAL, otherwise it is PROHIBITED.

These keywords are thus capitalized when used to unambiguously specify requirements over protocol and application features and behavior that affect the interoperability and security of implementations. When these words are not capitalized, they are meant in their natural-language sense.

- Strings that are to be taken literally are enclosed in "double quotes".
- Words that are emphasized are printed in *italic*.
- Data model names and values, and literal XML, are printed using the `data` character style.
- Keywords that are defined by the UPnP DM Working Committee are printed using the *forum* character style.
- Keywords that are defined by the UPnP Device Architecture are printed using the **arch** character style.
- A double colon delimiter, "::", signifies a hierarchical parent-child (parent::child) relationship between the two objects separated by the double colon. This delimiter is used in multiple contexts, for example: Service::Action(), Action()::Argument, parentProperty::childProperty.

1.4.1. Data Types

This specification uses data type definitions from two different sources. The UPnP Device Architecture defined data types are used to define state variable and action argument data types [UDA]. The XML Schema namespace is used to define XML-valued action arguments [XML-SCHEMA-2] (including the data model parameter values, see 2.3.2.1).

For UPnP Device Architecture defined Boolean data types, it is strongly RECOMMENDED to use the value "**0**" for false, and the value "**1**" for true. However, when used as input arguments, the values "**false**", "**no**", "**true**", "**yes**" may also be encountered and MUST be accepted. Nevertheless, it is strongly RECOMMENDED that all state variables and output arguments be represented as "**0**" and "**1**".

For XML Schema defined Boolean data types, it is strongly RECOMMENDED to use the value "*0*" for false, and the value "*1*" for true. However, when used within input arguments, the values "*false*", "*true*" may also be encountered and MUST be accepted. Nevertheless, it is strongly RECOMMENDED that all XML Boolean values be represented as "*0*" and "*1*".

XML elements that are of type `xsd:anySimpleType` (for example data model parameter values) MUST include an xsi:type attribute that indicates the actual data type of the element value. This is a SOAP requirement.

1.4.2. Strings Embedded in Other Strings

Some string variables, arguments and other XML elements and attributes (including data model parameter values) described in this document contains substrings that MUST be independently identifiable and extractable for other processing. This requires the definition of appropriate substring delimiters and an escaping mechanism so that these delimiters can also appear as ordinary characters in the string and/or its independent substrings.

This document uses such embedded strings in Comma Separated Value (CSV) lists (see section 1.5.1). Escaping conventions use the backslash character, "\" (character code U+005C), as follows:

a) Backslash ("\") is represented as "\\".

b) Comma (",") is represented as "\," in individual substring entries.

c) Double quote (“””) is not escaped.

This document also uses such embedded strings to represent XML documents (see section 1.5.2). Escaping conventions use XML entity references as specified in [XML] Section 2.4. For example:

a) Ampersand (“&”) is represented as “&” or via a numeric character reference.

b) Left angle bracket (“<”) is represented as “<” or via a numeric character reference.

c) Right angle bracket (“>”) usually doesn’t have to be escaped, but often is, in which case it is represented as “>” or via a numeric character reference.

1.5. Derived Data Types

This section defines a derived data type that is represented as a string data type with special syntax. This specification uses string data type definitions that originate from two different sources. The UPnP Device Architecture defined **string** data type is used to define state variable and action argument string data types. The XML Schema namespace is used to define `xsd:string` data types. The following definition applies to both string data types.

1.5.1. Comma Separated Value (CSV) Lists

The UPnP DM services use state variables, action arguments and other XML elements and attributes that represent lists – or one-dimensional arrays – of values. [UDA] does not provide for either an array type or a list type, so a list type is defined here. Lists MAY either be homogeneous (all values are the same type) or heterogeneous (values of different types are allowed). Lists MAY also consist of repeated occurrences of homogeneous or heterogeneous subsequences, all of which have the same syntax and semantics (same number of values, same value types and in the same order).

- The data type of a homogeneous list is **string** or `xsd:string` and denoted by CSV (x), where x is the type of the individual values.
- The data type of a heterogeneous list is also **string** or `xsd:string` and denoted by CSV (w, x [, y, z]), where w, x, y and z are the types of the individual values, and the square brackets indicate that y and z (and the preceding comma) are optional. If the number of values in the heterogeneous list is too large to show each type individually, that variable type is represented as CSV (*heterogeneous*), and the variable description includes additional information as to the expected sequence of values appearing in the list and their corresponding types. The data type of a repeated subsequence list is **string** or `xsd:string` and denoted by CSV ({w, x, y, z}), where w, x, y and z are the types of the individual values in the subsequence and the subsequence MAY be repeated zero or more times (in this case none of the values are optional).

The individual value types are specified as [UDA] data types or **A_ARG_TYPE** data types for **string** lists, and as [XML-SCHEMA-2] data types for `xsd:string` lists.

- A list is represented as a **string** type (for state variables and action arguments) or `xsd:string` type (within other XML elements and attributes).
- Commas separate values within a list.
- Integer values are represented in CSVs with the same syntax as the integer data type specified in [UDA] (that is: optional leading sign, optional leading zeroes, numeric ASCII).

- Boolean values are represented in state variable and action argument CSVs as either "**0**" for false or "**1**" for true. These values are a subset of the defined Boolean data type values specified in [UDA]: **0**, **false**, **no**, **1**, **true**, **yes**.
- Boolean values are represented in other XML element CSVs as either "*0*" for false or "*1*" for true. These values are a subset of the defined Boolean data type values specified in [XML-SCHEMA-2]: `0, false, 1, true`.
- Escaping conventions for the comma and backslash characters are defined in section 1.4.2.
- The number of values in a list is the number of unescaped commas, plus one. The one exception to this rule is that an empty string represents an empty list. This means that there is no way to represent a list consisting of a single empty string value.
- White space before, after, or interior to any numeric data type is not allowed.
- White space before, after, or interior to any other data type is part of the value.

Table 1-1: CSV Examples

Type refinement of string	Value	Comments
CSV (**string**)	"first,second"	List of 2 strings used as state variable or action argument value.
CSV (`xsd:string`)	"first,second"	List of 2 strings used within an XML element
CSV (`xsd:token`)	"first, second "	List of 2 strings used within an XML element. Each element is of type `xsd:token` so, even though the second value is " second " and has leading and trailing spaces, the value seen by the application will be "second" because `xsd:token` collapses whitespace.
CSV (**string**, **dateTime** [, **string**])	"Warning,2009-07-07T13:22:41, third\,value"	List of string, dateTime and (optional) string used as state variable or action argument value. Note the leading space and escaped comma in the third value, which is " third,value".
CSV (**string**, **dateTime** [, **string**])	"Warning,2009-07-07T13:22:41,"	As above but third value is empty.
CSV (**string**, **dateTime** [, **string**])	"Warning,2009-07-07T13:22:41"	As above but third value is omitted.

Type refinement of string	Value	Comments
CSV (*A_ARG_TYPE_Host*)	"grumpy,sleepy"	List of data items used as action argument value, each of which obeys the rules governing *A_ARG_TYPE_Host*. Any comma or backslash characters within a data item would have been escaped.
CSV (**i4**)	"1, 2"	Illegal CSV. White space is not allowed as part of an integer value.
CSV (**string**)	"a,,c,"	List of 4 strings "a", "", "c" and "".
CSV (**string**)	""	Empty list. It is not possible to create a list containing a single empty string.

1.5.2. Embedded XML Documents

An XML document is a string that represents a valid XML 1.0 document according to a specific schema. Every occurrence of the phrase "*XML Document*" is italicized and preceded by the document's root element name (also italicized), as listed in column 3, "Valid Root Element(s)" of Table 1-3, "Schema-related Information". For example, the phrase *SupportedDataModels XML Document* refers to a valid XML 1.0 document according to the CMS schema defined in Appendix A: XML schema (Normative). Such a document comprises a single `<SupportedDataModels …>` root element, optionally preceded by the XML declaration `<?xml version="1.0" …?>`.

This string will therefore be of one of the following two forms:

"`<SupportedDataModels …>…</SupportedDataModels>`"

or

"`<?xml …?><SupportedDataModels …>…</SupportedDataModels>`"

Escaping conventions for the ampersand, left angle bracket and right angle bracket characters are defined in section 1.4.2.

1.6. Management of XML Namespaces in Standardized DCPs

UPnP specifications make extensive use of XML namespaces. This allows separate DCPs, and even separate components of an individual DCP, to be designed independently and still avoid name collisions when they share XML documents. Every name in an XML document belongs to exactly one namespace. In documents, XML names appear in one of two forms: qualified or unqualified. An unqualified name (or no-colon-name) contains no colon (":") characters. An unqualified name belongs to the document's default namespace. A qualified name is two no-colon-names separated by one colon character. The no-colon-name before the colon is the qualified name's namespace prefix, the no-colon-name after the colon is the qualified name's "local" name (meaning local to the namespace identified by the namespace prefix). Similarly, the unqualified name is a local name in the default namespace.

The formal name of a namespace is a URI. The namespace prefix used in an XML document is not the name of the namespace. The namespace name is, or should be, globally unique. It has a single definition that is accessible to anyone who uses the namespace. It has the same meaning anywhere that it is used, both inside and outside XML documents. The namespace prefix, however, in formal XML usage, is defined only in an XML document. It must be locally unique to the document. Any valid XML no-colon-name may be used. And, in formal XML usage, no two XML documents are ever required to use the same namespace prefix to refer to the same namespace. The creation and use of the namespace prefix was standardized by the W3C XML Committee in [XML-NMSP] strictly as a convenient local shorthand replacement for the full URI name of a namespace in individual documents.

All of the namespaces used in this specification are listed in the Tables "Namespace Definitions" and "Schema-related Information". For each such namespace, Table 1-2, "Namespace Definitions" gives a brief description of it, its name (a URI) and its defined "standard" prefix name. Some namespaces included in these tables are not directly used or referenced in this document. They are included for completeness to accommodate those situations where this specification is used in conjunction with other UPnP specifications to construct a complete system of devices and services. The individual specifications in such collections all use the same standard prefix. The standard prefixes are also used in Table 1-3, "Schema-related Information", to cross-reference additional namespace information. This second table includes each namespace's valid XML document root element(s) (if any), its schema file name, versioning information (to be discussed in more detail below), and a link to the entry in Section 1.2 for its associated schema.

The normative definitions for these namespaces are the documents referenced in Table 1-3. The schemas are designed to support these definitions for both human understanding and as test tools. However, limitations of the XML Schema language itself make it difficult for the UPnP-defined schemas to accurately represent all details of the namespace definitions. As a result, the schemas will validate many XML documents that are not valid according to the specifications.

Table 1-2: Namespace Definitions

Standard Name-space Prefix	Namespace Name	Namespace Description	Normative Definition Document Reference
DM Working Committee defined namespaces			
cms	urn:schemas-upnp-org:dm:cms	CMS data structures	Appendix A: XML schema (Normative)
bmsnsl	urn:schemas-upnp-org:dm:bms:nsl	BMS NSLookupResult	[BMS]
Externally defined namespaces			
xsd	http://www.w3.org/2001/XMLSchema	XML Schema Language 1.0	[XML-SCHEMA-1] [XML-SCHEMA-2]
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema Instance Document schema	Sections 2.6 & 3.2.7 of: [XML-SCHEMA-1]
xml	http://www.w3.org/XML/1998/namespace	The "xml:" Namespace	[XML-NS]

Table 1-3: Schema-related Information

Standard Name-space Prefix	Relative URI and File Name[1] • Form 1, 2, 3	Valid Root Element(s)	Schema Reference
DM Working Committee defined namespaces			
cms	cms-vn-yyyymmdd.xsd cms-vn.xsd cms.xsd	<ContentPathList> <InstancePathList> <NodeAttributeValueList> <NodeAttrbutePathList> <ParameterInitialValueList> <ParameterValueList> <StructurePathList> <SupportedDataModels>	Appendix A: XML schema (Normative)
bmsnsl	bmsnsl-vn-yyyymmdd.xsd bmsnsl-vn.xsd bmsnsl.xsd	<NSLookupResult>	[BMS]

[1] Absolute URIs are generated by prefixing the relative URIs with "http://www.upnp.org/schemas/dm/".

1.6.1. Namespace Names, Namespace Versioning and Schema Versioning

The UPnP DM service specifications define several data structures (such as state variables and action arguments) whose format is an XML instance document that must comply with one or more specific XML namespaces. Each namespace is uniquely identified by an assigned namespace name. The namespaces that are defined by the DM Working Committee MUST be named by a URN. See Table 1-2 "Namespace Definitions" for a current list of namespace names. Additionally, each namespace corresponds to an XML schema document that provides a machine-readable representation of the associated namespace to enable automated validation of the XML (state variable or action parameter) instance documents.

Within an XML schema and XML instance document, the name of each corresponding namespace appears as the value of an xmlns attribute within the root element. Each xmlns attribute also includes a namespace prefix that is associated with that namespace in order to disambiguate (a.k.a. qualify) element and attribute names that are defined within different namespaces. The schemas that correspond to the listed namespaces are identified by URI values that are listed in the schemaLocation attribute also within the root element. (See Section 1.6.2)

In order to enable both forward and backward compatibility, namespace names are permanently assigned and MUST NOT change even when a new version of a specification changes the definition of a namespace. However, all changes to a namespace definition MUST be backward-compatible. In other words, the updated definition of a namespace MUST NOT invalidate any XML documents that comply with an earlier definition of that same namespace. This means, for example, that a namespace MUST NOT be changed so that a new element or attribute is required. Although namespace names MUST NOT change, namespaces still have version numbers that reflect a specific set of definitional changes. Each time the definition of a namespace is changed, the namespace's version number is incremented by one.

Each time a new namespace version is created, a new XML schema document (.xsd) is created and published so that the new namespace definition is represented in a machine-readable form. Since an XML schema document is just a representation of a namespace definition, translation errors can occur. Therefore, it is sometime necessary to re-release a published schema in order to correct typos or other namespace representation errors. In order to easily identify the potential multiplicity of schema releases for the same namespace, the URI of each released schema MUST conform to the following format (called Form 1):

Form 1: "http://www.upnp.org/schemas/dm/" ***schema-root-name*** "-v" ***ver*** "-" ***yyyymmdd***

where:

- ***schema-root-name*** is the name of the root element of the namespace that this schema represents.
- ***ver*** corresponds to the version number of the namespace that is represented by the schema.
- ***yyyymmdd*** is the year, month and day (in the Gregorian calendar) that this schema was released.

Table 1-3 "Schema-related Information" identifies the URI formats for each of the namespaces that are currently defined by the UPnP DM Working Committee.

As an example, the original schema URI for the "`cms`" namespace might be "http://www.upnp.org/schemas/dm/cms-v1-20091231.xsd". If the UPnP DM service specifications were subsequently updated in the year 2010, the URI for the updated version of the "`cms`" namespace might be "http://www.upnp.org/schemas/dm/cms-v2-20100906.xsd".

In addition to the dated schema URIs that are associated with each namespace, each namespace also has a set of undated schema URIs. These undated schema URIs have two distinct formats with slightly different meanings:

Form 2: "http://www.upnp.org/schemas/dm/" schema-root-name "-v" ver

Form 3: "http://www.upnp.org/schemas/dm/" schema-root-name

Form 2 of the undated schema URI is always linked to the most recent release of the schema that represents the version of the namespace indicated by ***ver***. For example, the undated URI "…/dm/cms-v2.xsd" is linked to the most recent schema release of version 2 of the "cms" namespace. Therefore, on September 06, 2010 (20100906), the undated schema URI might be linked to the schema that is otherwise known as "…/dm/cms-v2-20100906.xsd". Furthermore, if the schema for version 2 of the "cms" namespace was ever re-released, for example to fix a typo in the 20100906 schema, then the same undated schema URI ("…/dm/cms-v2.xsd") would automatically be updated to link to the updated version 2 schema for the "cms" namespace.

Form 3 of the undated schema URI is always linked to the most recent release of the schema that represents the highest version of the namespace that has been published. For example, on December 31, 2009 (20091231), the undated schema URI "…/dm/cms.xsd" might be linked to the schema that is otherwise known as "…/dm/cms-v1-20091231.xsd". However, on September 06, 2010 (20100906), that same undated schema URI might be linked to the schema that is otherwise known as "…/dm/cms-v2-20100906.xsd". When referencing a schema URI within an XML instance document or a referencing XML schema document, the following usage rules apply:

- All instance documents, whether generated by a service or a control point, MUST use Form 3.
- All UPnP DM published schemas that reference other UPnP DM schemas MUST also use Form 3.

Within an XML instance document, the definition for the schemaLocation attribute comes from the XML Schema namespace "http://www.w3.org/2002/XMLSchema-instance". A single occurrence of the attribute can declare the location of one or more schemas. The schemaLocation attribute value consists of a whitespace separated list of values that is interpreted as a namespace name followed by its schema location URL. This pair-sequence is repeated as necessary for the schemas that need to be located for this instance document.

In addition to the schema URI naming and usage rules described above, each released schema MUST contain a version attribute in the `<schema>` root element. Its value MUST correspond to the format:

ver "-" ***yyyymmdd*** where ***ver*** and ***yyyymmdd*** are described above.

The version attribute provides self-identification of the namespace version and release date of the schema itself. For example, within the original schema released for the "`cms`" namespace (…/cms-v1-20091231.xsd), the `<schema>` root element might contain the following attribute: `version="1-20091231"`.

1.6.2. Namespace Usage Examples

The schemaLocation attribute for XML instance documents comes from the XML Schema instance namespace "http:://www.w3.org/2001/XMLSchema-instance". A single occurrence of the attribute can declare the location of one or more schemas. The `schemaLocation` attribute value consists of a whitespace separated list of values: namespace name followed by its schema location URL. This pair-sequence is repeated as necessary for the schemas that need to be located for this instance document.

Example:

Sample CMS XML Instance Document. Note that the references to the UPnP DM schemas do not contain any version or release date information. In other words, the references follow Form 3 from above. Consequently, this example is valid for all releases of the UPnP DM service specifications.

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ParameterValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<Parameter>
<ParameterPath…</ParameterPath>
<Value>…</Value>
…
</cms:ParameterValueList>
```

1.7. Vendor Defined Extensions

In compliance with the UPnP Device Architecture approach, vendors MAY define their own extensions for this service to provide custom functionalities to devices.

Whenever vendors create additional vendor-defined state variables, actions or other XML elements and attributes, their assigned names and XML representation MUST follow the naming conventions and XML rules as specified in [UDA], Section 2.5, "Description: Non-standard vendor extensions".

The same "**X_**" rule described in [UDA] MUST be used whenever vendors create additional vendor-defined attributes, data types and so on. Their assigned names and XML representation MUST follow the naming conventions and XML rules as specified below:

- Attributes (see 2.3.2) supported by parameters can be extended adding vendor defined attributes. Such attributes MUST be named using the "**X_**" prefix as described above.
- Data types (see 2.3.2.1) can be extended adding vendor defined enumeration values, extending the list of possible values for parameters. All new enumeration values MUST be named using the "**X_**" prefix as described above.
- The data model (see 0 for further details) can be extended adding vendor defined parameters whereas their name must be defined using the using the "**X_**" prefix as described above. Vendors can also add subtrees anywhere in the supported data model adding new non standard *Nodes* named with the same "**X_**" rule. In this case, the contained *Nodes* are scoped by the parent node and do not need to be named using the "**X_**" rule.

2. Service Modeling Definitions

2.1. ServiceType

The following service type identifies a service that is compliant with this template:

urn:schemas-upnp-org:service:*ConfigurationManagement:1*.

2.2. Key Concepts

The CMS (*ConfigurationManagement:1* Service) manages configuration of *Parent Device* by means of actions that take effect on *Parent Device* parameters: the concept of *Parameter* is therefore at the center of CMS.

A *Parameter* has two basic properties:

- Its **name**, that uniquely identifies a *Parameter* managed by CMS actions.
- Its **value**, that represent the actual value of the *Parameter* read from the *Parent Device* or to be set on the *Parent Device* by CMS actions.

The parameters that a *Parent Device* supports are defined with the concept of *data model*, basically a hierarchical set of unique *Parameter* names with the associated *Parameter* definition properties (e.g. syntax type, description, default value, allowed values). This specification of CMS defines a basic data model including a set of mandatory and optional parameters; other parameters not defined in the CMS basic data model may be provided by the *Parent Device*, as additional parameters defined by other UPnP DCPs or as vendor specific parameters. The definition of such data model extensions could be imported from other UPnP Working Committees (e.g. a data model defined by the UPnP AV WC) or other organizations (e.g. BBF STBService defined in TR-135 from the BroadBand Forum). Refer to Appendix C: Mapping rules for Other for further details.

Parameters in the data model also have **attributes** containing additional information about the parameters. Examples of **attributes** are the type (e.g.: the *Parameter* is a string, a number or something else), the Access rules (e.g.: the *Parameter* is read only or read write) and so on.

Parameter names, *Parameter* values and *Paramter* attributes are exchanged among the control point and the *Parent Device* using input and output action arguments. Their syntax is XML based on an XSD defined in this specification. *Parameter* names' syntax used in the XML fragments is defined using EBNF-style grammar.

2.2.1. Data Model Management Basics

Parameters in the data model (see Appendix B: Common Objects) are modeled using a hierarchical structure like a logical tree, quite similar to directories and files in a file system. The control point can read and write their values by specifying a name that uniquely identifies the *Parameter*. The CMS actions defined in this UPnP service type reference *parameters* with the "name-value pair" approach, i.e.:

- When reading *parameters*, the control point sends to the *Parent Device* a request with a list of *Parameter* names to be read from the *Parent Device*, and the *Parent Device* responds to the control point with a list of pairs of {*Parameter* name; *Parameter* value}.
- When writing *parameters*, the control point sends to the *Parent Device* a request with a list of pairs of {*Parameter* name; *Parameter* value} to be changed on the *Parent Device*.

- When creating object instances the control point sends to the *Parent Device* a request with the name of the parent of the object to be created and the *Parent Device* responds with the object instance identifier. The control point can then initialize the *parameters* in the new object instance by passing to the *Parent Device* a list of pairs of {*Parameter* name; *Parameter* value} to be configured.
- When deleting object instances the control point sends to the *Parent Device* a request with the name of the object to be deleted and the *Parent Device* will remove the object instance, all its *parameters*, and any sub-objects.

2.3. Syntax for *Parameter* Names

Various *Parent Device* management actions need to handle *Nodes* in the data model tree. Thus, in order to specify the input and output arguments of these actions, an appropriate syntax is necessary. This section describes the glossary of basic terms and the syntax.

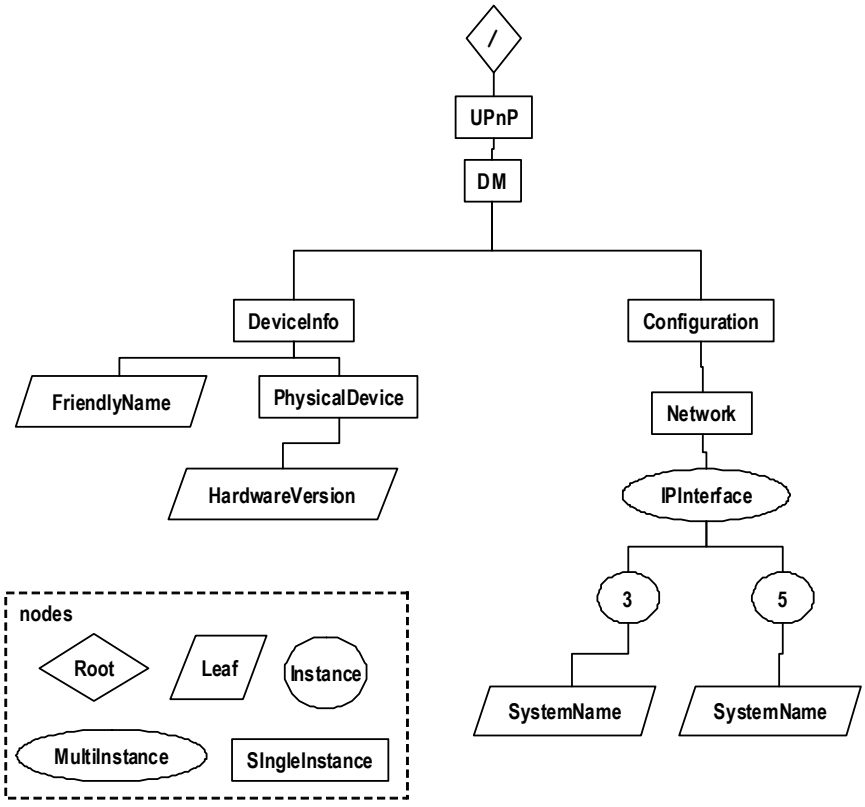

Figure 1: example of structured tree excerpted from the CMS data model.

Examples in this section are taken from the data model defined in Appendix B: Common Objects and from [SMS] when necessary.

2.3.1.1. Definition of Terms

Because of its hierarchical nature, the data model can be represented as a logical tree of *Nodes*. The relationship between two consecutive and connected *Nodes* is a parent-child relationship.

Below there is the list of the terms used to describe the *Nodes* and their structure:

Node: This represents any element of the data model tree. A *Node* may have a parent *Node* as well as children *Nodes*. All *Nodes* have a name, and each *Node* can be uniquely identified by a sequence of *Nodes* (in a parent to child relationship) from the origin (i.e. the *Root* of the tree) to that specific *Node*. The different kinds of *Nodes* such as listed below:

> *Root*: this is a special *Node* in the data model tree because all other *Nodes* are descendant of the *Root Node*. The *Root* has no parent *Node*. The *Root* is always identified by the name / (the slash symbol).
>
> *Leaf*: this kind of *Node* has a parent *Node* but does not have children *Nodes*. A specific property of *Leaf Nodes* is that they have an associated value.
>
> *SingleInstance*: this is an intermediate *Node* which has one parent and may have one or more named children *Nodes* forming a sub-tree below this *Node*.
>
> *MultiInstance*: this is a special intermediate *Node* which can contain a collection of *Instance Nodes* (in the same way a table contains rows).
>
> *Instance*: this *Node* represents a sort of table row belonging to the parent *MultiInstance Node*. This table row (which is indeed a sub-tree of the data model) can be created at run-time and added as an instance to the *MultiInstance Node*. The *Instance Node* can also be dynamically deleted as well.

Path: is a **string representation** of the sequence of *Nodes* starting with the *Root Node* and ending at the *Node* of interest. Specifically it's the concatenation of the *Node* names. Due to the tree structure of the data model, a *Path* from the *Root* to a *Node* is unique.

*Parameter***:** the *Parameter* is a piece of information in the data model and is identified by its name which is a **fully qualified name** starting from the *Root Node*, passing by static or dynamically created intermediate *Nodes*, and ending to the *Leaf Node* (which is therefore uniquely identified) that contains actual value: **only** *Parameters* **have values**. The parameter name is the corresponding *Leaf Node*'s path. For example, in terms of *Path*, the *Parameter* name is equivalent to a *Path* from the *Root* to the *Leaf*.

Some parameters are read-only (i.e. the control point can only read their values) and some others are writable (i.e. the control point can both read and change their values).

Figure 1 shows an example hierarchy from the data model. There is the *Root Node* / that includes all other *Nodes* in the tree. The `DeviceInfo` is a *SingleInstance* containing another *SingleInstance* `Capabilities` and a *Leaf* `FriendlyName`. The `IPInterface` *Node* is a *MultiInstance* containing two instances identified with the numbers `3` and `5`. Each instance of the `IPInterface` *MultiInstance Node* has the same content: a *Leaf* named `SystemName`. The complete list of parameters represented in Figure 1 is:

```
/UPnP/DM/DeviceInfo/FriendlyName
/UPnP/DM/DeviceInfo/PhysicalDevice/HardwareVersion
/UPnP/DM/Configuration/Network/IPInterface/3/SystemName
/UPnP/DM/Configuration/Network/IPInterface/5/SystemName
```

The following list is an example of all possible path types:

```
/UPnP/DM/DeviceInfo/FriendlyName                    /* root */
                                                    /* following are paths from root …*/
/UPnP/                                                /* … to SingleInstance node*/
/UPnP/DM/                                             /* …to SingleInstance node */
/UPnP/DM/DeviceInfo/FriendlyName                      /* …to Leaf node */
/UPnP/DM/DeviceInfo/PhysicalDevice/                   /* …to SingleInstance node */
/UPnP/DM/DeviceInfo/PhysicalDevice/HardwareVersion    /* …to Leaf node */
```

```
/UPnP/DM/Configuration/                                  /* …to SingleInstance node*/
/UPnP/DM/Configuration/Network/                          /* …to SingleInstance node*/
/UPnP/DM/Configuration/Network/IPInterface/              /* …to MultiInstance node*/
/UPnP/DM/Configuration/Network/IPInterface/3/            /* …to Instance node 3*/
/UPnP/DM/Configuration/Network/IPInterface/3/SystemName /* …to Leaf node*/
/UPnP/DM/Configuration/Network/IPInterface/5/            /* …to Instance node 5*/
/UPnP/DM/Configuration/Network/IPInterface/5/SystemName /* …to Leaf node*/
```

2.3.1.2. Definition of Grammar

In order to represent the parameters from the structured data model tree into the flat XML fragment of action arguments, the following EBNF-style syntax [EBNF] grammar is defined.

The grammar described herein is normative and is defined in the XML schema: Appendix A: XML schema (Normative).

The grammar can be used **to match** a sequence of characters in order to verify whether it corresponds to a syntactically correct sequence of *Nodes* from the *Root* to a *Node* or symmetrically **to produce** a syntactically correct sequence of character which corresponds to a sequence of *Nodes* from the *Root* to a *Node*. Parent-child relationship between *Nodes* is represented in the sequence of character by the "/" symbol between the parent *Node* name (on the left side of the "/") and the child *Node* name (on the right side of the "/").

The grammar defined below is organized in four set of rules. The first set contains rules for the basic syntactical definitions named "**Basic matching rules**". Then there is a short set named "**Auxiliary rules**" with internal definitions. The third set is named "**Matching rules for specific types of paths**" and contains specific rules for the basic terms defined below.

The fourth set is named "**Matching rules for composite paths**" and contains rules (i.e. *PartialPath*, *ContentPath*, *StructurePath* and *ParameterInitializationPath*) to define the syntax for paths whereas the most of them are a choice of a combination of the path types defined above. Such rules are needed to provide the strongest **type checking** as possible for action arguments, defined via A_ARG_TYPE_ state variables.

```
/* Basic matching rules */
Alpha       ::= [a-zA-Z]
Numeric     ::= [0-9] | [1-9][0-9]+
SpecialChar ::= "_"
Wildchar    ::= "#"

NodeName  ::= NCName /* as defined in [XML-NCName], see
                     "Restrictions to NCName" in the text below. */
LeafName  ::= NodeName

SingleInstanceNodeName ::= NodeName "/"
MultiInstanceNodeName  ::= NodeName "/"
Instance               ::= Numeric "/"
InstanceAlias          ::= Wildchar "/"

/* Auxiliary rules */

InternalNode ::= SingleInstanceNodeName |
                 MultiInstanceNodeName Instance
InternalAlias ::= SingleInstanceNodeName |
                  MultiInstanceNodeName InstanceAlias


/* Matching rules for specific types of paths */
RootPath           ::= "/"
```

```
ParameterPath       ::= RootPath InternalNode* LeafName
SingleInstancePath  ::= RootPath | RootPath InternalNode*
SingleInstanceNodeName
MultiInstancePath   ::= RootPath InternalNode* MultiInstanceNodeName
InstancePath        ::= RootPath InternalNode* MultiInstanceNodeName
                        Instance

/* Matching rules for composite paths */
PartialPath         ::= RootPath |
                        SingleInstancePath |
                        MultiInstancePath |
                        InstancePath
ContentPath         ::= PartialPath | ParameterPath
StructurePath       ::= RootPath InternalAlias* LeafName?
ParameterInitializationPath      ::= SingleInstanceNodeName* LeafName
```

Basic Matching Rules

Restrictions to NCName: the NCName in [XML-NCName] leads to a large number of possible characters that can be used for *Node* names. Due to some constraints in data models from other organizations (see Appendix C: Mapping rules for Other Organizations) the "." and "-" characters MUST NOT be used.

Matching Rules for Specific Types of Paths

- *RootPath*: to define the syntax for the *Root Node*. *RootPath* always matches/produce the "/".
- *SingleInstancePath*: to define the syntax for a path starting from the *Root Node* and ending with a *SingleInstance Node*. Therefore *SingleInstancePath* always defines paths ending with a NodeName (which is a *SingleInstance Node*) followed by the "/" symbol. *SingleInstancePath* and *MultiInstancePath* (defined below) are syntactically identical. *SingleInstancePath* is used, for example, when retrieving the values of all its descendants using a single *GetValues()* action invocation.
- *MultiInstancePath*: to define the syntax for a path starting from the *Root Node* and ending with a *MultiInstance Node*. Therefore *MultiInstancePath* always defines paths ending with a NodeName (which is a *MultiInstance Node*) followed by the "/" symbol. *SingleInstancePath* (defined above) and *MultiInstancePath* are syntactically identical. *MultiInstancePath* is used, for example, when creating a new *Instance Node* using the *CreateInstance()* action.
- *InstancePath*: to define the syntax for a path starting from the *Root Node* and ending with a *Instance Node* (a *MultiInstancePath* followed by an *Instance Node* name). Therefore *InstancePath* always defines paths ending with a *Node* name (which is an *Instance Node*) followed by the "/" symbol. *InstancePath* is used, for example, to delete an existing *Instance* using the *DeleteInstance()* action.
- *ParameterPath*: to define the syntax for a path starting from the *Root Node* and ending with a *Leaf Node*: which is the fully qualified name for the *Parameter*. *ParameterPaths* are used, for example, in the name-value pairs when setting the value of a *Parameter*.

Matching Rules for Composite Paths

- *PartialPath*: is a path from the *Root* to a *Node* in the data model tree which is not a *Leaf*. *PartialPath* is indeed either a *RootPath* or a *SingleInstancePath* or a *MultiInstancePath* or an *InstancePath*. The partial path always ends with a slash symbol. *PartialPath* is used in *A_ARG_TYPE_PartialPath* state variable. Examples of *PartialPaths* are:

```
/
/UPnP/DM/Configuration/Network/
/UPnP/DM/Configuration/Network/IPInterface/
/UPnP/DM/Configuration/Network/IPInterface/2/
```

```
/UPnP/DM/Configuration/Network/IPInterface/5/IPv4/
```

- *ContentPath*: is a path from the *Root* to a *Node* in the data model tree which can be either the *Root* or a *SingleInstance Node* or a *MultiInstance Node* or a *Instance Node* or a *Leaf*. In other words the *ContentPath* can be either a *PartialPath* or a parameter (i.e. *ParameterPath*) and include all *Node* types except the `InstanceAlias`. *ContentPath* is used in *A_ARG_TYPE_ContentPathList* state variable.

- *ParameterInitializationPath*: is a sequence of *Nodes* starting from *SingleInstance Node* and ending to the *Leaf Node*. In other words it is a *ParameterPath* which starts from a *SingleInstance Node* rather than from the *Root Node*. The sequence of *SingleInstance Nodes* on the left of the *Leaf Node* can be empty. This *ParameterInitializationPath* is specifically used in *A_ARG_TYPE_ParameterInitialValueList*. *ParameterInitializationPaths* do not begin with the "/". Examples of valid *ParameterInitializationPaths* are:

```
SystemName
IPv4/IPAddress
IPv4/AddressingType
AddressingType
```

- *StructurePath*: is a path from the *Root* to a *Node* which includes (in case *Instance Nodes* are included in the path) the wild-chars # instead of table *Instances*, that are therefore forbidden. *StructurePath* is used when browsing the actual data model tree, hence the wild-char # means "every instances" that could belong to the *MultiInstance Node*. A valid *StructurePath* can end with a wild-char, a *SingleInstance Node* or *Leaf Node*. Due to the *StructurePath* syntax, when no *Instance Node* is included in the path, a *PartialPath* or even a *ParameterPath* are also *StructurePaths*. *StructurePath* is used in *A_ARG_TYPE_StructurePath* and *A_ARG_TYPE_StructurePathList* state variables. Examples of *StructurePaths* are:

```
/
/UPnP/DM/DeviceInfo/
/UPnP/DM/DeviceInfo/PhysicalDevice/HardwareVersion
…
/UPnP/DM/Configuration/Network/Interface/#/
/UPnP/DM/Configuration/Network/Interface/#/IPv4/IPAddress
```

2.3.2. Attributes

Attributes are used to specify properties of *Nodes*, such as, for example, the data type of a *Leaf* or the access permission to create a new instance of a *MultiInstance Node*.

Values of attributes are managed using CMS actions in the same way that *Parameters* are: the XML fragments in specific actions' arguments carry the attribute values.

There are two types of attributes:

- **ReadOnly**: the attribute value is specified in the data model definition and cannot be explicitly changed by the control pointduring the lifetime of the *Parent Device*.

- **ReadWrite**: the attribute value is up to the *Parent Device* implementation and can be dynamically changed by the control point using the *SetAttributes()* action, see section 2.6.10. When the control point changes the value of one or more attributes, an event associated with the *AttributeValuesUpdate* state variable is generated. This is because, for example, other control points have to be informed if they potentially will not receive any more change notifications for

some *Parameter* they are interested in. Data model definitions may contain default values for ReadWrite attributes.

For the purposes of CMS the following attributes are defined for *Nodes*:

Table 2-4: *Nodes* attributes

Name	Type	Value type	Values
Type	*ReadOnly*	*String*	*string,* *int,* *unsignedInt,* *long,* *unsignedLong,* *boolean,* *dateTime,* *base64,* *hexBinary*
Access	*ReadOnly*	*String*	*readWrite,* *readOnly*
Version	*ReadOnly*	*unsignedInt*	*0,1,2,…*
EventOnChange	*ReadWrite*	*Boolean*	*1,* *0.*
MIMEType	*ReadOnly*	*String*	*(see section 2.3.2.5)*
Non-standard attributes implemented by an UPnP vendor go here.	*TBD*	*TBD*	*TBD*

Depending on *Node* types, attributes can be required (Req.), optional (Opt.) or not applicable (N/A), and have different meaning. If the attribute is required means that its value MUST be returned using the *GetAttributes()* action and their default values MUST be specified in the data models.

Table 2-5: Requirements for attributes

Node	*Type*	*Access*	*Version*	*Event-OnChange*	*MIMEType*
Root	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Leaf	*Req.*	*Req.*	*Opt.*	*Req.*	*Opt.*
SingleInstance	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
MultiInstance	*N/A*	*Req.*	*Opt.*	*Req.*	*N/A*

Node	Type	Access	Version	Event-OnChange	MIMEType
Instance	*N/A*	*Req.*	*N/A*	*N/A*	*N/A*

Attributes supported by parameters can be extended adding vendor defined attributes, as described in 1.7.

2.3.2.1. Type

This REQUIRED attribute describes the *Parameters* type, making use of a limited subset of the SOAP data types (see Appendix B: Common Objects).

In the case of data model extensions, each vendor/organization is then responsible for defining its own rules to integrate new data model definitions. Rules refer to syntax renaming (see Appendix C: Mapping rules for Other) and type conversion for [SOAP] encoding.

For some numerical types (e.g.: int, long, …), a value range may be given using the form <type>[Min:Max], where the Min and Max values are inclusive. If either Min or Max are missing, this indicates no limit. For example, unsignedInt[3:] means all valid 4 bytes unsigned integers from 3 to 4294967295. A "k" or "K" suffix is interpreted as a 1024 (not 1000) multiplier, e.g. 32k means 32768.

For types expressed as subset of the ISO 8601 (e.g. dateTime and Time stamps in this specification) used to describe relative time since reboot, the value MUST be expressed in UTC (Universal Coordinated Time) unless explicitly stated otherwise in the definition of a parameter of this type. If absolute time is not available to the *Parent Device*, it SHOULD instead indicate the relative time since boot, where the boot time is assumed to be the beginning of the first day of January of year 1, or 0001-01 -01T00:00:00. For example, 2 days, 3 hours, 4 minutes and 5 seconds since boot would be expressed as 0001-01-03T03:04:05. Relative time since boot MUST be expressed using an untimezoned representation. Any untimezoned value with a year value less than 1000 MUST be interpreted as a relative time since boot. If the time is unknown or not applicable, the following value representing "Unknown Time" MUST be used: 0001-01 -01T00:00:00Z.

Table 2-6: *Type* attribute values description

Type	Description
string	*Unicode string. For strings listed in this specification, a minimum and maximum allowed length can be listed using the form string(Min:Max), where Min and Max are the minimum and maximum string length in characters. If either Min or Max are missing, this indicates no limit, and if Min is missing the colon can also be omitted, as in string(Max). Multiple comma-separated ranges can be specified, in which case the string length MUST be in one of the ranges. A "k" or "K" suffix is interpreted as a 1024 (not 1000) multiplier, e.g. 32k means 32768.* *For strings in which the content is an enumeration, the longest enumerated value ilmplicitly determines the maximum length.* *When transporting a string value within an XML document, any characters which are special to XML MUST be escaped as specified by the XML specification [*[XML]*]. Additionally, any characters other than printable ASCII characters, i.e. any characters whose decimal ASCII representations are outside the (inclusive) ranges 9-10 and 32-126, SHOULD be escaped as specified by the XML specification.*
int	*Integer in the range –2147483648 to +2147483647, inclusive. See the introductory text for details on range specifications.*
long	*Long integer in the range –9223372036854775808 to 9223372036854775807, inclusive. See the introductory text for details on range specifications.*
unsignedInt	*Unsigned integer in the range 0 to 4294967295, inclusive. See the introductory text for details on range specifications.*

Type	Description
unsignedLong	*Unsigned long integer in the range 0 to 18446744073709551615, inclusive. See the introductory text for details on range specifications.*
boolean	*Boolean, where the allowed values are "0", "1", "true", and "false". The values "1" and "true" are considered interchangeable, where both equivalently represent the logical value true. Similarly, the values "0" and "false" are considered interchangeable, where both equivalently represent the logical value false.* *It is STRONGLY RECOMMENDED to use "0" and "1".*
dateTime	*The subset of the ISO 8601 date-time format defined by the SOAP dateTime type.* *interpreted as a relative time since boot (see introductory text for more details on usage of ISO 8601 date-time format).*
base64	*Base64 encoded binary (no line-length limitation).* *A minimum and maximum allowed length can be listed per string types using the form base64(Min:Max), where Min and Max are the minimum and maximum length in characters before Base64 encoding. If either Min or Max are missing, this indicates no limit, and if Min is missing the colon can also be omitted, as in base64(Max). Multiple commaseparated ranges can be specified, in which case the length MUST be in one of the ranges. A "k" or "K" suffix is interpreted as a 1024 (not 1000) multiplier, e.g. 32k means 32768.*
hexBinary	*Hex encoded binary.* *A minimum and maximum allowed length can be listed per string using the form hexBinary(Min:Max), where Min and Max are the minimum and maximum length in characters before Hex Binary encoding. If either Min or Max are missing, this indicates no limit, and if Min is missing the colon can also be omitted, as in hexBinary(Max). Multiple commaseparated ranges can be specified, in which case the length MUST be in one of the ranges. A "k" or "K" suffix is interpreted as a 1024 (not 1000) multiplier, e.g. 32k means 32768.*

All IPv4 addresses and subnet masks MUST be represented as strings in IPv4 dotted-decimal notation. All IPv6 addresses and subnet masks MUST be represented using any of the 3 standard textual representations as defined in [RFC 3513], sections 2.2.1, 2.2.2 and 2.2.3. Both lower-case and upper-case letters can be used. Use of the lower-case letters is RECOMMENDED. Examples of valid IPv6 address textual representations:

- 1080:0:0:800:ba98:3210:11aa:12dd
- 1080::800:ba98:3210:11aa:12dd
- 0:0:0:0:0:0:13.1.68.3

Unspecified or inapplicable IP addresses and subnet masks MUST be represented as empty strings unless otherwise specified by the parameter definition.

All MAC addresses are represented as strings of 12 hexadecimal digits (digits 0-9, letters A-F or a-f) displayed as six pairs of digits separated by colons. Unspecified or inapplicable MAC addresses MUST be represented as empty strings unless otherwise specified by the parameter definition.

In case of enumeration parameters, which have string type, new enumeration values can be added by vendors. In compliance with the UPnP Device Architecture approach, enumeration values that are defined as vendor proprietary extensions must begin with the prefix **X_**.

The value of *Type* attribute MUST be specified in each data model definition.

2.3.2.2. Access

The *Access* attribute is REQUIRED and is used to specify whether a control point can or not change the value of a *Parameter* as well as create and delete an *Instance Node*, therefore:

- Read/Write access for a parameter. In this case it is associated with a *Leaf Node*.
- Read/Write access for a *MultiInstance* and *Instance Nodes*. Read access means the *Instance Nodes* of the *MultiInstance Node* can only be addressed by reading actions. Concerning Write access:
  - The argument of *CreateInstance()* action is a *MultiInstance Node*, therefore a *MultiInstance Node* having *Write access* means that *Instances* can be created. This attribute has to be specified in the *MultiInstance Nodes* of the data model.
  - The argument of *DeleteInstance()* action is an *Instance Node*, therefore an *Instance Node* having Write access means that *Instances* can be deleted. This attribute has to be specified in the *Instance Nodes* of the data model.
  - *Instance Nodes* may have different access attribute value in comparison with their *MultiInstance* as it is explained in the table below.

Possible values for this attribute are *readOnly* and *readWrite*.

In case a parameter needs to be written and not to be read (e.g. a typical example is a parameter which is a password), it is suggested to the data models' designers to specify that, when read password values the empty string might be returned instead of a read error.

The data model definition specifies the "highest" right *Access* to a parameter, but right applies at run-time may be more restrictive. For example, a data model definition might not specify any right restriction but a implementation can enforce a *readOnly* permission.

Table 2-7: *Access* Attribute Semantics

Node	Value	Description
Root	*readWrite*	*N/A*
	readOnly	*N/A*
Leaf	*readWrite*	*The value of the parameter associated with this Leaf Node can be read using the GetValues() and GetSelectedValues() actions and can be written using the SetValues() action.*
	readOnly	*The value of the parameter associated with this Leaf Node can be read using the GetValues() and GetSelectedValues() actions. If the control point attempts a write operation on the Parameter using the SetValues() action the Parent Device returns an error and the action fails.*
SingleInstance	*readWrite*	*N/A*
	readOnly	*N/A*
MultiInstance	*readWrite*	*New Instance Nodes can be created using the CreateInstance() action.*
	readOnly	*An attempt to create a new Instance Node using the CreateInstance() action fails and the Parent Device returns an error.*
Instance	*readWrite*	*An existing Instance Node can be deleted using the DeleteInstance() action.*
	readOnly	*An attempt to delete an existing instance using the DeleteInstance() action fails and the Parent Device returns an error.*

Leaf and *MultiInstance Nodes* having *readOnly* *Access* attribute value are completely under control of the *Parent Device*, therefore there is no way for the control point to change their values using CMS actions.

The value of *Access* attribute MUST be specified in each data model definitions.

2.3.2.3. EventOnChange

This attribute has effect on the *ConfigurationUpdate* state variable. It is associated to some *Leaf* and *MultiInstance Nodes* and indicates whether the *ConfigurationUpdate* must be updated and therefore the corresponding event must be generated.

This attribute is REQUIRED if the data model specification requires events on change of paramters'value. Indeed, the *Common Objects* specified in this service as well as data models specified in other UPnP services and data models specified elsewhere (e.g. vendor extension data models or data models defined by other organizations) must define whether a *Leaf* or a *MultiInstance Node* MUST support the *EventOnChange* attribute. It is up to the implementation to support this attribute also for other *Leaf* or a *MultiInstance Nodes* whereas the *EventOnChange* is not explicitly required in the data model.

The *EventOnChange* attribute value for a *Parameter* is not related to the *Access* attribute value of the same parameter. Therefore, *readOnly Parameters* can also support the *EventOnChange* attribute.

The *EventOnChange* attribute value MUST be persistent hence the CMS must maintain its value when disappears from the network and reappears again later sending the **ssdp::alive** message. Therefore, after the service reappears on the network, the control point will receive the notification on change for the same *Parameters* unless:

- Another control point has changed the attribute values in the meantime, or
- One or more software modules containing the implementation of such *Parameters* was removed or replaced with a new one, or
- The entire *Parent Device* firmware has been changed.

The following table defines the semantics for *Nodes* which implement the attribute. Refer to *ConfigurationUpdate* (see section 2.4.1) state variable and the relationship between state variables (section 2.4.22) for further details.

The default value of *EventOnChange* attribute SHOULD be specified in data model definitions; otherwise default values are implementation specific.

Table 2-8: *EventOnChange* Attribute Semantics

Node	Value	Description
Root	*1*	*N/A*
	0	*N/A*
Leaf	*1*	*If the value of the parameter associated with this Leaf Node changes its value the ConfigurationUpdate state variable must be updated and therefore an event must be sent to the subscribed CPs.*
	0	*If the value of the parameter associated with this Leaf Node changes its value the ConfigurationUpdate state variable must not be updated.*
SingleInstance	*1*	*N/A*

Node	Value	Description
	0	*N/A*
MultiInstance	*1*	*If a new Instance Node for this MultiInstance Node is created the ConfigurationUpdate state variable must be updated and therefore an event must be sent to the subscribed CPs.* *If an existing Instance Node for this MultiInstance Node is deleted the ConfigurationUpdate state variable must be updated and therefore an event must be sent to the subscribed CPs.*
	0	*If a new Instance Node for this MultiInstance Node is created the ConfigurationUpdate state variable must not be updated.* *If an existing Instance Node for this MultiInstance Node is deleted the ConfigurationUpdate state variable must not be updated.*
Instance	*1*	*N/A*
	0	*N/A*

2.3.2.4. Version

This OPTIONAL attribute may be used to keep track on data model value changes (*Parameter* value change and/or instance creation/deletion). The *Version* is an attribute specific for *Leaf Nodes* and *MultiInstance Nodes*. Whenever a *Parameter* changes its value or an *Instance* of a *MultiInstance Node* is created or deleted, the associated *Version* attribute assumes the new value of the *CurrentConfigurationVersion* state variable. Since multiple changes are possible; i.e., that more than a single *Parameter* is changed using the same *SetValue()* action whether to group multiple changes in a single update of the *CurrentConfigurationVersion* implementation dependent..

The *Version* attribute value MUST be persistent, hence the CMS must maintain its value when disappears from the network and reappears again later sending the **ssdp::alive** message.

The version attribute can therefore be used for version control; i.e., *Nodes* which support the version attribute could be considered as under version control.

If the *Version* attribute is supported, the data model specifies for which *Nodes* it is mandatory. *Nodes* which have *Version* attribute are considered under version control.

The data model specified in this service and in other services which support CMS and data model extensions defines a minimum list of *Parameters* (specifically *Leaf* and *MultiInstance Nodes*) which must support the *Version* attribute, when the *Version* attribute is implemented by the *Parent Device*. In case the *Version* attribute is supported, no partial implementation is permitted concerning the list of *Parameters*: all the required ones from the specification must have the *Version* attribute support or none have the *Version* attribute supported. The control point can use the *GetAttributes()* to know whether a parameter or a *MultiInstance Node* supports the attribute.

The following table summarizes the *Version* attribute semantics. Refer to the *CurrentConfigurationVersion* (see section 2.4.2) and the relationship between state variables section (section 2.4.22) for further details.

Table 2-9: *Version* Attribute Semantics

Node	Description
Root	*N/A*
Leaf	*If the value of the parameter associated with this Leaf Node changes its value the Version attribute value assumes the same value of the CurrentConfigurationVersion state variable.*
SingleInstance	*N/A*
MultiInstance	*Version attribute value assumes the same value of the CurrentConfigurationVersion state variable when:* *- A new Instance Node for this MultiInstance Node is created,* *- An existing Instance Node for this MultiInstance Node is deleted.*
Instance	*N/A*

The following example clarifies how the *Version* attribute may be implemented by the *Parent Device*, in order to realize the expected behavior. Suppose that 0 is the starting value for *CurrentConfigurationVersion* and three *Nodes* (supporting the *Version* attribute) are involved: node_1, node_2 and node_3. As the first *Node* is modified, the example sequence starts:

- Step 1: node_2 is modified, hence

  CurrentConfigurationVersion is updated to 1,

  Version(node_1) is left unchanged at its starting value 0,

  Version(node_2) is set to *CurrentConfigurationVersion* value, so its value becomes 1,

  Version(node_3) is left unchanged to its default starting value 0.

- Step 2: node_3 is modified, hence

  CurrentConfigurationVersion is updated to 2,

  Version(node_1) is left unchanged at its starting value 0,

  Version(node_2) is set left unchanged to its previous value 1,

  Version(node_3) is set to *CurrentConfigurationVersion* value, so its value become 2.

- Step 3: node_2 is modified once again, hence

  CurrentConfigurationVersion is updated to 3,

  Version(node_1) is left unchanged at its starting value 0,

  Version(node_2) is set to *CurrentConfigurationVersion* value, so its value become 3,

  Version(node_3) is set left unchanged to its previous value 2.

2.3.2.5. MIMEType

This OPTIONAL attribute describes the MIME type for *Parameters* whose *Type* attribute value is **string**. MIME is a standardized way of describing the type of content in a file. It is composed of 2 parts, a type and a subtype. The *MIMEType* attribute, when supported, must be associated to *Parameters* only, hence it applies to *Leaf Nodes*.

Standard values for this attribute are defined in [IANA-MIME]. Example *MIMEType* valid values are:

```
application/pdf
text/plain,
text/xml,
text/html,
audio/3gpp
image/jpeg
video/mpeg
video/mp4 etc.
video/MP4V-ES
```

Vendor extensions are permitted by providing more values for such attribute. Since the *MIMEType* is application oriented, the generic control point can ignore the syntax and meaning values for such attribute and treat them as is was a simple string of characters.

2.3.3. *Instance Nodes* as Primary Keys and Unique Keys Extension

Instance Node names, which are unsigned integers, are the **primary key** to uniquely identify sub-tree instances of a *MultiInstance Node* in the data model. The syntax of instance *Nodes* has been defined in section 2.3.1.2. This means that a control point is able to address a specific instance in the data model when reading or writing some of its children *Nodes*. For example, the *Parameter*:

`/UPnP/DM/Configuration/Network/IPInterface/15/IPv4/IPAddress`

addresses the `IPAddress` *Leaf Node* which is contained in the *Instance* number 15 within the *MultiInstance* `Interface`. Therefore the number 15 is the value of the primary key for this *Instance*.

As an additional and OPTIONAL feature to address instances, the *Parent Device* MAY offer **the unique key** extension. **Unique keys** allow the control point to address instances using value of specific *Leaf Nodes* rather than using instance numbers only, therefore **unique keys** uniquely identify instances.

In case the *Parent Device* implements the **unique key** it MUST support the following extension to the grammar:

```
Instance           ::= Numeric "/" | UniqueKey "/"
UniqueKey          ::= "{" UniqueKeyMatches "}"
UniqueKeyMatches   ::= UniqueKeyMatch |
                       UniqueKeyMatch ";" UniqueKeyMatches
UniqueKeyMatch     ::= ParameterInitializationPath "=" ParameterValue
ParameterValue     ::= /* The value to be compared. It must be a valid
                          literal for the data type, and strings must
                          be escaped. */
```

As it is defined in the grammar above, **unique keys** may be composed by one or more *Parameter* as it must be specified in the data model. This means that in case the *Parent Device* supports the **unique key** addressing, the vendor must specify in the data model which are the *Leaf Nodes* contained in the *MultiInstance Node* that are used to make the **unique key**.

For example, given again the following parameter instanced in the data model:

`/UPnP/DM/Configuration/Network/IPInterface/15/IPv4/IPAddress`

Supposing its value is "`239.255.255.250`" whereas the value of

`/UPnP/DM/Configuration/Network/IPInterface/15/SystemName`

within the same Interface instance is "`AdvertisementInterface`". The *Parent Device* might offer another way to address the IPAddress *Parameter*. Indeed, if the *Parent Device* also supports **unique keys**, and the `SystemName` is defined as **unique key**, the control point may also use the following syntax to address the same *Parameter*:

`/UPnP/…/IPInterface/{SystemName="AdvertisementInterface"}/IPv4/IPAddress`

The **unique key** addressing is an extension and MUST NOT replace the basic primary key addressing using the *Instance Node*.

In order to guarantee backward compatibility for control points which does not support such extended addressing mechanism, if the control point does not make use of **unique keys** in action arguments (i.e. it uses the primary key addressing), the *Parent Device* MUST not use **unique keys** in the responses (i.e. it must use the primary key addressing).

In case this **unique key** extension is supported by the device, the data model of the device MUST specify in its description which parameters are **unique keys** for a specific *MultiInstance Node*.

This syntax extension for primary keys MUST be supported by *Parent Devices* when they import data models which make use of non numeric values to identify *Instance Nodes*;i.e., wherever the data model does not use a device-assigned unsigned integer to identify object instances (see: Appendix C: Mapping rules for Other …).

2.3.4. Time stamps

Time stamps are used in this specification, specifically in the CSV strings used in some state variables to inform the CPs about some relevant event. Valid values for time stamps are defined in section 1.4.1.

2.4. State Variables

Unlike most other services, the *ConfigurationManagement:1* service is primarily *Node*-based as described above. The service state variables exist to support argument passing in the service actions. Information is not exposed directly through explicit state variables. Rather, a client retrieves *ConfigurationManagement:1* service information via the return arguments of the actions defined below.

Reader Note: For a first-time reader, it may be more helpful to read the action definitions before reading the state variable definitions.

Table 2-1: State Variables

Variable Name	Req. or Opt.[1]	Data Type	Allowed Value [2]	Default Value [2]	Eng. Units
ConfigurationUpdate	*R*	**string**	CSV(**ui4**, **dateTime** [, **string**]) list.See section 2.4.1	""	
CurrentConfigurationVersion	*R*	**ui4**	See section 2.4.2	*0*	
SupportedDataModelsUpdate	*R*	**string**	CSV(**ui4**, **dateTime** [, **string**]) list.See section 2.4.3	""	
SupportedParametersUpdate	*R*	**string**	CSV(**ui4**, **dateTime** [, **string**]) list.See section 2.4.4	""	
AttributeValuesUpdate	*O*	**string**	CSV(**ui4**, **dateTime** [, **string**]) list.See section 2.4.5	""	
InconsistentStatus	*O*	**boolean**	**0**, **1**. See section 2.4.6	**0**	
A_ARG_TYPE_StructurePath	*R*	**string**	Formatted string. See section 2.4.7	""	
A_ARG_TYPE_StructurePathList	*R*	**string**	XML string. See section 2.4.8	See section 2.4.8	
A_ARG_TYPE_PartialPath	*R*	**string**	Formatted string. See section 0	""	
A_ARG_TYPE_ParameterValueList	*R*	**string**	XML string. See section 2.4.10	See section 2.4.10	

[1] R = Required, O = Optional, X = Non-standard.

[2] Values listed in this column are required. To specify standard optional values or to delegate assignment of values to the vendor, you must reference a specific instance of an appropriate table below.

Variable Name	Req. or Opt. [1]	Data Type	Allowed Value [2]	Default Value [2]	Eng. Units
A_ARG_TYPE_NodeAttributeValueList	*R*	**string**	XML string. See section 2.4.11	See section 2.4.11	
A_ARG_TYPE_ParameterInitialValueList	*R*	**string**	XML string. See section 2.4.12	See section 2.4.12	
A_ARG_TYPE_Filter	*R*	**string**	Formatted string. See section 2.4.13	""	
A_ARG_TYPE_SupportedDataModels	*R*	**string**	XML string. See section 2.4.14	See section 2.4.14	
A_ARG_TYPE_SearchDepth	*R*	**ui4**	See section 2.4.15	0	
A_ARG_TYPE_ChangeStatus	*R*	**string**	*ChangesCommitted*, *ChangesApplied* See section 2.4.16	See section 2.4.16	
A_ARG_TYPE_InstancePathList	*R*	**string**	XML string. See section 2.4.17	See section 2.4.17	
A_ARG_TYPE_ContentPathList	*R*	**string**	XML string. See section 2.4.18	See section 2.4.18	
A_ARG_TYPE_MultiInstancePath	*R*	**string**	Formatted string. See section 2.4.19	See section 2.4.19	
A_ARG_TYPE_InstancePath	*R*	**string**	Formatted string. See section 2.4.20	*""*	
A_ARG_TYPE_NodeAttributePathList	*R*	**string**	XML string. See section 2.4.21	See section 2.4.21	
Non-standard state variables implemented by an UPnP vendor go here.	*X*	*TBD*	*TBD*	*TBD*	*TBD*

2.4.1. ConfigurationUpdate

The *ConfigurationUpdate* state variable is REQUIRED. It keeps track of changes of all *Nodes* under version control; refer to the *Version* attribute (see section 2.3.2.4) for further details. It is a CSV (**ui4**, **dateTime** [, **string**]) list (1.5.1), where:

- The first element of the CSV is the last value of *CurrentConfigurationVersion* state variable.
- The second element of the CSV is the time stamp when the *CurrentConfigurationVersion* changed its value. Refer to section 2.3.4 for time stamp requirements.
- The control point must ignore what is returned in this CSV from the third element on, after the last trailing comma. The last trailing comma is not required.

Example of valid *ConfigurationUpdate* is the following string:

```
356,2007-10-24T05:41:00
```

where the `356` is the value of *CurrentConfigurationVersion* and the `2007-10-24T05:41:00` is the time stamp when the *CurrentConfigurationVersion* changed its.

The value of *ConfigurationUpdate* MUST be persistent and survive as the CMS disappears from the network and reappears again later sending the **ssdp::alive** message. It is evented at a maximum rate of 5 Hz (once every 0.2 seconds).

Refer to the section 2.4.22 for further details.

2.4.2. CurrentConfigurationVersion

The *CurrentConfigurationVersion* state variable is REQUIRED. *CurrentConfigurationVersion* is of type **ui4**, starting from 0. It is incremented by one each time the value of a *Leaf* or *MultiInstance Node* supporting the *Version* attribute **changes**.

Changes in the *Parent Device* configuration are defined as following:

- The value of a *Parameter* (value associated to a *Leaf Node*) in the supported data model is changed because of the *SetValues()* action or some event that is outside of the UPnP scope, for example an external event like a user action (such as via the GUI) on the *Parent Device*.
- An *Instance Node* is created or deleted in the supported data model because of *CreateInstance()* or *DeleteInstance()* actions or some event that is outside of the UPnP scope, for example an external event like a user action (wuch as via the GUI) on the *Parent Device*. For example, if a *MultiInstance Node* is under version control, each time a new *Instance Node* is created or an existing one is deleted, the *CurrentConfigurationVersion* is incremented by 1.
- It is implementation specific whether each single change in the configuration *Parameters* leads to an increment or multiple value changes can be grouped to cause a single change in *CurrentConfigurationVersion*. For example, if *SetValues()* action invocation is used to change the value of 3 different *Parameters*, it is an implementation choice to define whether the *CurrentConfigurationVersion* is:
  - incremented by 1 (one per action invocation), or
  - incremented by 3 (one per *Parameter* value changed).

The value of the *Version* attribute for each *Parameter* must be updated accordingly with the implemented behavior. From the example above:

- If the *CurrentConfigurationVersion* is incremented by 1 (one per action invocation), the *Parameters*' *Version* attributes will have the same value, otherwise

- If the *CurrentConfigurationVersion* is incremented by 3 (one per *Parameter* value changed), each *Parameter* will have a different *Version* attribute value. How the *CurrentConfigurationVersion* values are assigned to *Parameters*' *Version* attribute values is an implementation choice.

Actions that fail not cause any configuration state change, and therefore the *CurrentConfigurationVersion* does not change.

When the maximum value of the **ui4** type is reached, the sequence is restarted from 0.

Refer to the section 2.4.22 for further details.

2.4.3. *SupportedDataModelsUpdate*

The *SupportedDataModelsUpdate* state variable is REQUIRED and keeps track of any changes in the supported data models (see section 2.4.14). This state variable allows a control point to know if there is a change in the list of supported data models as a result of firmware/software changes in the *Parent Device* as well as other external eventswhich are out of the scope of this service specification. *SupportedDataModelsUpdate* is a CSV (**ui4**, **dateTime** [, **string**]) list (1.5.1) where:

- The first element of the CSV is a sequential counter that is incremented by 1 whenever there is a change in the supported data model list,
- The second element of the CSV is the time stamp when the sequential counter changed its value. Refer to section 2.3.4 for time stamp's requirements.
- The control point must ignore what is returned in this CSV from the third element on, after the last trailing comma. The last trailing comma is not required

Example of valid *SupportedDataModelsUpdate* is the following string:

```
35,2008-10-24T05:45:30
```

where the `35` is the value of the sequential counter and the `2008-10-24T05:45:30` is the time stamp when the sequential counter changed its value.

This variable is evented and the event is moderated at a maximum rate of 1 Hz (once every 1.0 seconds).

The *SupportedDataModelsUpdate* MUST be persistent and survive as the CMS disappears from the network and reappears again later sending the **ssdp::alive** message.

2.4.4. *SupportedParametersUpdate*

The *SupportedParametersUpdate* state variable is REQUIRED and keeps track of any changes in the list of supported *Parameters* of the data models supported by the *Parent Device*. This state variable allows a control point to know if there is a change on the list of the *Parent Device* supported *Parameters*, triggered by events out of the scope of this service specification like, for example, a firmware change, software modules change or end-user interaction. *SupportedParametersUpdate* is a CSV (**ui4**, **dateTime** [, **string**]) list (1.5.1), where:

- The first element of the CSV is a sequential counter that is incremented by 1 whenever there's a change in the supported *Parameters*,
- The second element of the CSV is the time stamp when the sequential counter changed its value. Refer to section 2.3.4 for time stamp's requirements.
- The control point must ignore what is returned in this CSV from the third element on, after the last trailing comma. The last trailing comma is not required

Example of valid *SupportedParametersUpdate* is the following string:

```
59,2008-10-24T05:45:30
```

where the `59` is the value of the sequential counter and the `2008-10-24T05:45:30` is the time stamp when the sequential counter changed its value.

This variable is evented and the event is moderated at a maximum rate of 1 Hz (once every 1.0 seconds).

The *SupportedParametersUpdate* MUST be persistent and survive as the CMS disappears from the network and reappears again later sending the **ssdp::alive** message.

2.4.5. *AttributeValuesUpdate*

The *AttributeValuesUpdate* state variable is OPTIONAL and keeps track of any changes in the attribute values for *Parameters* in the data models supported by the *Parent Device*. This state variable allows a control point to know if there is a change on some attribute values due to:

- *SetAttributes()* action invocation (i.e., changes in attribute values from another control point),
- Some event (some could be external and out of the scope of this service) causing some changes in the supported data model and therefore in the attribute values (e.g.: a firmware change, software modules change or end-user interaction (such as via a GUI)).

AttributeValuesUpdate is a CSV (**ui4**, **dateTime** [, **string**]) list (1.5.1), where:

- The first element of the CSV is a sequential counter that is incremented by 1 whenever there's a change in the attribute values,
- The second element of the CSV is the time stamp when the sequential counter changed its value. Refer to section 2.3.4 for time stamp's requirements.
- The control point must ignore what is returned in this CSV from the third element on, after the last trailing comma. The last trailing comma is not required

Example of valid *AttributeValuesUpdate* is the following string:

```
59,2008-10-24T05:45:30
```

where the `59` is the value of the sequential counter and the `2008-10-24T05:45:30` is the time stamp when the sequential counter changed its value.

This variable is evented and the event is moderated at a maximum rate of 1 Hz (once every 1.0 seconds).

The *AttributeValues* MUST be persistent and survive as the CMS disappears from the network and reappears again later sending the **ssdp::alive** message.

2.4.6. *InconsistentStatus*

The *InconsistentStatus* state variable is OPTIONAL and keeps track whether the *Parent Device* configuration is consistent or not. As the control point uses *SetValues()*, *CreateInstance()*, *DeleteInstance()* or *SetAttributes()* action to change the configuration of the *Parent Device*, the *Parent Device* MAY use the *Status* argument (see the *A_ARG_TYPE_ChangeStatus* for further explanations) to return information about its internal status, concerning the consistency and the need to perform further operation (e.g.: a reboot of the operating system supporting this CMS) in order to apply all the changes.

Table 2-2: allowedValueList for *InconsistentStatus*

Value	Req. or Opt.	Description
1	R	The *InconsistentStatus* is set to 1 when the control point uses *SetValues()*, *CreateInstance()*, *DeleteInstance()* or *SetAttributes()* action and the Status argument value returned is *ChangesCommitted*. The InconsistentStatus may be also autonomously set to 1 by the *Parent Device* when the same internal condition occurs, due to some event which is out of the scope of UPnP DM. The default value for Inconsistent status is 0 because as the *Parent Device* starts and therefore sends the **ssdp::alive** message, its internal status MUST be consistent.
0	R	The *InconsistentStatus* state variable is set back to its default value of 0 as soon as the status is once again consistent (e.g.: all pending changes have been applied). It's up to the implementation to return to a consistent status (e.g. apply the changes) as soon as possible, and the status MUST be consistent whenever CMS is announced via **ssdp::alive** messages.

InconsistentStatus is a global information of the *Parent Device*, whereas the A_ARG_TYPE_ChangeStatus returned by *SetValues()*, *CreateInstance()*, *DeleteInstance()* and *SetAttributes()* actions invocation is a local information strictly related to the action behavior. Therefore the A_ARG_TYPE_ChangeStatus returned by subsequent action invocations are not related one to each other.

2.4.7. *A_ARG_TYPE_StructurePath*

This state variable (defined for the purpose of specifying an action argument) represents a *StructurePath*. This means it must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *StructurePath*.

2.4.8. *A_ARG_TYPE_StructurePathList*

This state variable (defined for the purpose of specifying an action argument) represents a list of StructurePaths. This means it must be correctly validated using the XML schema in Appendix A: XML schema. Each element of the list must be correctly parsed (i.e., syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *StructurePath*. The specific portion of the schema to be considered is the one starting with the element named *StructurePathList*.

The following XML file shows an *A_ARG_TYPE_StructurePathList* example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<cms:StructurePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<StructurePath>/UPnP/DM/DeviceInfo/</StructurePath>
<StructurePath>/UPnP/DM/DeviceInfo/FriendlyName</StructurePath>
<StructurePath>/UPnP/DM/DeviceInfo/PhysicalDevice/NetworkInterface/#/</S
tructurePath>
</cms:StructurePathList>
```

In case the list of *StructurePaths* returned contains no elements, the valid XML file MUST be anyway returned as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:StructurePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd"/>
```

2.4.9. *A_ARG_TYPE_PartialPath*

This state variable (defined for the purpose of specifying an action argument) represents a *PartialPath*. This means it must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *PartialPath*.

2.4.10. *A_ARG_TYPE_ParameterValueList*

This state variable (defined for the purpose of specifying an action argument) represents a list of pairs *ParameterPath*-value. This means it must be correctly validated using the XML schema in Appendix A: XML schema. Each `<ParameterPath>` element of the list must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *ParameterPath*. The specific portion of the schema to be considered is the one starting with the element named *ParameterValueList*.

The following XML file shows an *A_ARG_TYPE_ParameterValueList* example:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ParameterValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd”>
<Parameter>
<ParameterPath>/UPnP/DM/Configuration/Network/IPInterface/15/SystemName<
/ParameterPath>
<Value>AdvertisementInterface</Value>
</Parameter>
<Parameter>
<ParameterPath>/UPnP/DM/Configuration/Network/IPInterface/15/IPv4/IPAddr
ess</ParameterPath>
<Value>239.255.255.250</Value>
</Parameter>
</cms:ParameterValueList>
```

In case the list of `ParameterPath-Value` pairs returned contains no elements, the valid XML file MUST be anyway returned, as:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<cms:ParameterValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd"/>
```

2.4.11.A_ARG_TYPE_NodeAttributeValueList

This state variable (defined for the purpose of specifying an action argument) represents a list composed of either a *ParameterPath*, a *MultiInstancePath* or an *InstancePath* associated with one or more *Parameter* elements (<Type>, <Access> and so on: see section 2.3.2). This means it must be correctly validated using the XML schema in Appendix A: XML schema. Each <AttributePath> element of the list must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from respectively the grammar rules named *ParameterPath*, *MultiInstancePath* or *InstancePath*. The specific portion of the schema to be considered is the one starting with the element named *NodeAttributeValueList*.

The following XML file shows an *A_ARG_TYPE_NodeAttributeValueList* example:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:NodeAttributeValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<Node> <!-- ParameterPath-->
<NodeAttributePath>/UPnP/DM/DeviceInfo/FriendlyName</NodeAttributePath>
<Type>string</Type>
<Access>readWrite</Access>
</Node>

<Node> <!-- MultiInstancePath -->
<NodeAttributePath>/UPnP/DM/DeviceInfo/PhysicalDevice/Interface/</NodeAt
tributePath>
<Type>MultiInstance</Type>
<Access>readOnly</Access>
</Node>

<Node> <!-- InstancePath -->
<NodeAttributePath>/UPnP/DM/Configuration/Network/Interface/3/</NodeAttr
ibutePath>
<Type>Instance</Type>
<Access>readOnly</Access>
</Node>
</cms:NodeAttributeValueList>
```

In case the list of *Parameters* returned contains no elements, the valid XML file MUST be anyway returned, as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:NodeAttributeValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd"/>
```

2.4.12.A_ARG_TYPE_ParameterInitialValueList

This state variable (defined for the purpose of specifying an action argument) represents a specific XML fragment used to initialize children *Nodes* of a *MultiInstance Node* when creating a new *Instance* in the *Parent Device* (i.e. the *Instance* to be created is therefore not yet known by the control point). In other words, it allows the control point to indicate the initial values of the new Node in an efficient manner

during *MultiInstance Node* creation.This state variable, when instanced in the proper action, must be correctly validated using the XML schema in Appendix A: XML schema. The specific portion of the schema to be considered is the one starting with the element named ParameterInitialValueList. The XML element named *ParameterInitializationPath* must be correctly matched/produced using the grammar in section 2.3.1.2 starting from the proper grammar rule named *ParameterInitializationPath*. Such *ParameterInitializationPath* list is used to initialize what is content within the *Instance* to be created: the *ParameterInitializationPath* is needed because the *Leaf* to be initialized could be contained in a *SingleInstance Nodes* (or a sequence of nested ones) instead of being a direct child of the *Instance Node* to be created.

There is no *MultiInstance Node* which is creatable in CMS. For the purposes of this example to explain the syntax of the *A_ARG_TYPE_ParameterInitialValueList* state variable, the following *MultiInstance Node* is considered as it was creatable (i.e. as it had readWrite value for Access attribute):

```
/UPnP/DM/Configuration/Network/IPInterface/
```

If the control point needs to create a new instance of the *MultiInstance Node* above, and needs to initialize at the same time the value of its child:

```
/UPnP/DM/Configuration/Network/IPInterface/#/IPv4/IPAddress
```

The following XML fragment must be used:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ParameterInitialValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<Node>
<ParameterInitializationPath>IPv4/IPAddress</ParameterInitializationPath
>
<Value>239.255.255.250</Value>
</Node>
</cms:ParameterInitialValueList>
```

2.4.13. *A_ARG_TYPE_Filter*

This state variable is defined for the purpose of describing the *GetSelectedValues()* action argument and is used to reduce the size of the action response with a basic filtering functionality. There are some situations where, for example, the number of *Instance Nodes* is quite large and the control point is really interested only in retrieve some particular *Nodes* rather than reading all instances with *GetInstances()* or *GetValues()*. A filter is formed by a predicate on the value of a given *Parameter*.

Filter strings syntax is described here formally using an EBNF-style grammar [EBNF] and is an extension of the given grammar for *Parameters* (see section 2.3.1.2).

```
Filter     ::= 1 | Cond (LogOp Cond)*
Cond       ::= ValueComparison |
               ParametersComparison |
               AttributeComparison
ValueComparison      ::= StructurePath RelOp ParameterValue
ParametersComparison ::= StructurePath RelOp ParameterPath
AttributeComparison  ::= AttributeName RelOp AttributeValue

AttributeName  ::= "Version"
```

```
AttributeValue ::= /* the value to be compared must be a valid type for
the AttributeName specified.*/
Numeric        ::= /* as defined in section 2.3.1.2 */
ParameterPath  ::= /* as defined in section 2.3.1.2 */
RelOp          ::= "<" | "<=" | "=" | "!=" | ">" | ">="
ParameterValue ::= /* the value to be compared must be a valid literal
for the data type, and strings must be quoted -> the string must be
escaped because they could contain some special chars.*/
LogOp          ::= 'and' | 'or'

/*************************************************************/
/*        Operator precedence and associativity              */
/* Listed in order of precendence (highest:40 to lowest:10)  */
/*                                                           */
/* precedence       operator        associativity            */
/* 40               <,<=,>,>=       left-to-right            */
/* 30               =,!=            left-to-right            */
/* 20               and             left-to-right            */
/* 10               or              left-to-right            */
/*                                                           */
/*************************************************************/
```

Examples of filters from the [SMS] data model.

To retrieve the list of *Parameters* whereas the State of the DU is either *Unresolved* or *Installing*:

```
/UPnP/DM/Software/DU/#/State = "Unresolved" or
/UPnP/DM/Software/DU/#/State = "Installing"
```

To retrieve the list of *Parameters* whereas the EUID is equal to 145:

```
/UPnP/DM/Software/DU/#/EU/#/EUID = 145
```

To retrieve the list of *Parameters* whereas the DUType is equal to “Firmware”:

```
/UPnP/DM/Software/DU/#/DUType = "Firmware"
```

The filter can also be used, when the *Version* attribute is implemented by the *Parent Device*, to retrieve *Parameters* that have a specific value (or range of values) for that attribute.

For example, in case the control point receives an event due to the *ConfigurationUpdate* changes to 2395, if the control point needs to know which are the *Parameters* changed their value correspondingly with the *ConfigurationUpdate* event, it must query the *Parent Device* with *GetSelectedValues()* action using the filter:

Version = 2395.

For backwards compatibility, if the *Parent Device* does not implement the *AttributeComparison* grammar rule it MUST ignore such filtering condition assuming a logical “true” as result. *AttributeComparison* grammar rule may be extended by *Parent Device* implementations because of the support for vendor specific attributes.

2.4.14. *A_ARG_TYPE_SupportedDataModels*

This state variable (defined for the purpose of specifying an action argument) represents a specific XML fragment used to define the table of the *Parent Device*’s supported data models. This state variable, when instanced in the action *GetSupportedDataModels()*, must be correctly validated using the XML schema in

Appendix A: XML schema. The XML elements must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the proper grammar rule named. The specific portion of the schema to be considered is the one starting with the element named SupportedDataModels.

The SupportedDataModels table has the following columns:

- **URI:** (REQUIRED) the URI indicates the following attributes of the supported data model: (a) the organization that defined it, (b) the specification in which it is defined, and (c) the version of the specification. URI format rules are specified independently for each organization. This URI relates only to the organization and the specification and does NOT indicate which part of the specification is supported by the *Parent Device*.
- **Location:** (REQUIRED) is a *SingleInstancePath* identifying the attachment point of the supported data model into the *Parent Device* data model Locations in the SupportedDataModels table need not be unique in order to let the same mounting point be used for different data models supported. Therefore given a Location for a supported data model, all the *Parameter* of such supported data model MUST have the same Location as a prefix starting from the *Root Node*.
- **URL:** (OPTIONAL) refers to a resource that describes which parts of the specification are supported. URL format rules, and rules governing the referenced resource, are specified independently for each organization. Regardless of whether the URL is supplied the *GetSupportedParameters()* and *GetAttributes()* actions can return basic information about the supported data model. The URL can provide a mechanism suitable for CPs to retrieve more detailed information.
- **Description:** (OPTIONAL) informative description of the supported data model.
- **SourceLocation:** (OPTIONAL) is the path from the *Root* of the imported data model to the *Node* that is to be attached to Location with respect to the document where the data model is defined in the external location. The SourceLocation can be either a fully qualified path (i.e. a *Path* from the *Root Node*) or a relative path. If the SourceLocation is a fully qualified path the Location can be the empty string, otherwise the Location is the prefix to add to this SourceLocation to build the fully qualified path.

The unique key for the SupportedDataModels table is the couple of the required elements (URI,Location), in order to uniquely identify each rows (i.e. instances of `SubTree`).

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:SupportedDataModels xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
  <SubTree>
    <URI>
      urn:UPnP:Parent Device:1:ConfigurationManagement:1
    </URI>
    <Location>/UPnP/DM/CMS/</Location>
    <Description>
      UPnP Manageable Device common objects for CMS</Description>
  </SubTree>
  <SubTree>
    <URI>
      urn:UPnP:Parent Device:1:SoftwareManagement:1
    </URI>
    <Location>/UPnP/DM/Software/</Location>
    <Description>
      UPnP Manageable Device common objects for SMS
    </Description>
```

```
  </SubTree>
  <SubTree>
    <URI>
      urn:broadband-forum-org:tr-135-1-0-0
    </URI>
    <Location>/BBF/STBService/</Location>
    <URL>http://www.example.com/upnp/stb/bbf-stb-1-0.xml</URL>
    <Description>TR-135 STBService Object</Description>
  </SubTree>
  <SubTree>
    <URI>
      urn:ietf:rfc:3729
    </URI>
    <Location>/IETF/MIB/APM/</Location>
    <Description>RFC 3729 APM-MIB</Description>
  </SubTree>
  <SubTree>
    <URI>
      urn:Manufacturer:spec_v1.html
    </URI>
    <Location>/UPnP/DM/CMS/DeviceInfo/X_CustomInfo/</Location>
    <URL>http://www.example.com/Manufacturer/spec_v1.xml</URL>
    <Description>Vendor extension</Description>
  </SubTree>
</cms:SupportedDataModels>
```

2.4.15.A_ARG_TYPE_SearchDepth

This state variable (defined for the purpose of specifying an action argument) represents the depth of the search for the *GetSupportedParameters()* and *GetInstances()* actions, in terms of number of traversed *Nodes*, where each *Node* traversed represents a single level of depth. The usage of this argument is specified in the actions' descriptions.

2.4.16.A_ARG_TYPE_ChangeStatus

This state variable (defined for the purpose of specifying an action argument) represents the status of the requested changes after one of the following action is performed: *SetValues()*, *SetAttributes()*, *CreateInstance()* or *DeleteInstance()*.

Table 2-3: allowedValueList for *A_ARG_TYPE_ChangeStatus*

Value	Req. or Opt.	Description
ChangesCommitted	*R*	All changes required by the action have been validated and committed but some or all are not yet applied (for example, if a reboot of the underlying operating system is necessary before the new values are applied).
ChangesApplied	*R*	All changes required by the action have been validated, committed and applied.

It is strongly RECOMMENDED that devices implementations apply changes as they are requested by the control point and therefore return *ChangesApplied* rather than only commiting and leaving the device in

an inconsistent status. The exception to this recommendation is when the device delays applying changes because of the control point's use of *BMS::SetSequenceMode()* as descrbed below.

When the *Parent Device* returns the *ChangesCommitted* value to the control point it means that the internal status may be not completely consistent because of some further internal operations need to be executed before the status will return consistent. For example the new values have been saved somewhere but the *Parent Device* does not currently use them and an autonomous reboot is required in order to let the *Parent Device* read the new values and use them. In the opposite situation the *Parent Device* returns *ChangesApplied* because it starts immediately using the new values for the running configuration.

It is not REQUIRED for the *Parent Device* to use both values: if the *Parent Device* is able to apply all changes immediately it will use the *ChangesApplied* value only. And this is the desired approach for all devices implementations.

The status returned by the *Parent Device* could also be affected by the *BMS::SetSequenceMode()* [BMS] value. In case the *BMS::SequenceMode* is **1**, a smart *Parent Device* MAY delay the application of changes until the *BMS::SequenceMode* values will return to **0** therefore it might return *ChangesCommitted* (instead of the *ChangesApplied*) during this phase.

2.4.17.*A_ARG_TYPE_InstancePathList*

This state variable (defined for the purpose of specifying an action argument) represents a list of *InstancePaths*. This means it must be correctly validated using the XML schema in Appendix A: XML schema. Each element of the list must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *InstancePaths*. The specific portion of the schema to be considered is the one starting with the element named *InstancePathList*.

The following XML file shows an *A_ARG_TYPE_InstancePathList* example as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:InstancePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<InstancePath>
/UPnP/DM/Configuration/Network/Interface/5/
</InstancePath>
</cms:InstancePathList>
```

In case the list of *InstancePaths* returned contains no elements, the valid XML file MUST be anyway returned, as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:InstancePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd"/>
```

2.4.18.*A_ARG_TYPE_ContentPathList*

This state variable (defined for the purpose of specifying an action argument) represents a list of *ContentPaths*. This means it must be correctly validated using the XML schema in Appendix A: XML schema. Each element of the list must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *ContentPaths*, therefore they could be *RootPath*,

SingleInstancePaths, *MultiInstancePaths*, *InstancePaths* or *ParameterPaths*. The specific portion of the schema to be considered is the one starting with the element named *ContentPathList*.

The following XML file shows an *A_ARG_TYPE_ContentPathList* example as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ContentPathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<!-- RootPath-->
<Path>/</Path>

<!-- SingleInstancePath-->
<ContentPath>/UPnP/DM/DeviceInfo/</ContentPath>

<!--  MultiInstancePath -->
<ContentPath>/UPnP/DM/DeviceInfo/PhysicalDevice/Interface/</ContentPath>

<!-- InstancePath -->
<ContentPath>/UPnP/DM/Configuration/Network/Interface/3/</ContentPath>

<!-- ParameterPath -->
<ContentPath>/UPnP/DM/Configuration/Network/Interface/15/IPv4/IPAddress<
/ContentPath>
</cms:ContentPathList>
```

In case the list of *ContentPaths* returned contains no elements, the valid XML file MUST be anyway returned, containing as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ContentPathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd"/>
```

2.4.19. A_ARG_TYPE_MultiInstancePath

This state variable (defined for the purpose of specifying an action argument) represents a *MultiInstancePath*. This means it must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *MultiInstancePath*.

2.4.20. A_ARG_TYPE_InstancePath

This state variable (defined for the purpose of specifying an action argument) represents an *InstancePath*. This means it must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rule named *InstancePath*.

2.4.21. A_ARG_TYPE_NodeAttributePathList

This state variable (defined for the purpose of specifying an action argument) represents a list of *ParameterPaths* mixed with *MultiInstancePaths* and *InstancePaths*, because attributes are related to them.

This state variable must be correctly validated using the XML schema in Appendix A: XML schema. Each element of the list must be correctly parsed (i.e. syntactically produced) using the grammar in section 2.3.1.2 starting from the grammar rules named *ParameterPath*, *MultiInstancePath* or *InstancePath*. The specific portion of the schema to be considered is the one starting with the element named *NodeAttributePathList*.

The following XML file shows an *A_ARG_TYPE_NodeAttributePathList* example as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:NodeAttributePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">

<!-- ParameterPath-->
<NodeAttributePath>/UPnP/DM/DeviceInfo/FriendlyName</NodeAttributePath>

<!--  MultiInstancePath -->
<NodeAttributePath>/UPnP/DM/DeviceInfo/PhysicalDevice/Interface/</NodeAt
tributePath>

<!-- InstancePath -->
<NodeAttributePath>/UPnP/DM/Configuration/Network/Interface/3/</NodeAttr
ibutePath>
</cms:NodeAttributePathList>
```

In case the list returned contains no elements, the valid XML file MUST be anyway returned, as:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:NodeAttributePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd"/>
```

2.4.22.Relationships Between State Variables

The *SupportedDataModelsUpdate*, *SupportedParametersUpdate, ConfigurationUpdate* and *AttributeValuesUpdate* state variables may be related one to each other (e.g. changes in the data model supported can have side effects on the *Parameters*' attribute values, although this is not required to be the case). Therefore it is up to the device to manage dependencies amongst these variables and generate events properly depending on the implementation.

The value of the *InconsistentStatus* conditionally depends from the *A_ARG_TYPE_ChangeStatus* value returned by the *Parent Device* when the *A_ARG_TYPE_ChangeStatus* returned is *ChangesCommitted*; if the action causes internal inconsistencies because changes have not yet been applied, it can lead to inconsistency at the global level.

The relationship and the sequence of internal operations between the *ConfigurationUpdate*, the *CurrentConfigurationVersion* and the attributes *EventOnChange* and *Version* are explained in the following diagrams.

If the *Node* does not support the *EventOnChange* attribute, the *ConfigurationUpdate* must not be updated and therefore no event must be sent as the *Node* value changes.

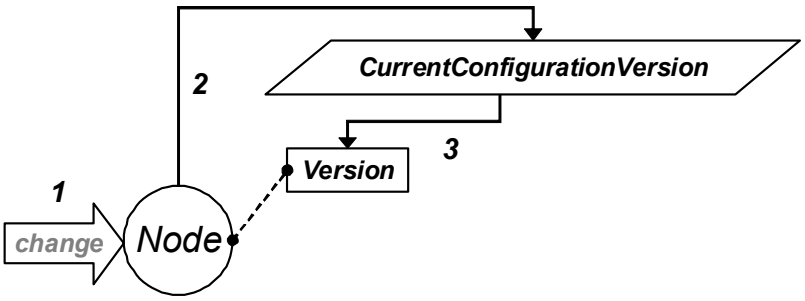

Figure 2: sequence from the *Version* attribute perspective.

The Figure 2 shows the sequence of operations in case the *Version* attribute only is supported by the *Node*. Internal steps as the *Node* value changes are the following:

1. A change occurs to the *Node*, due to an action execution or some other event out of the UPnP protocol scope.
2. If the *Node* supports the *Version* attribute, the *CurrentConfigurationVersion* must be updated (increased).
3. The *Version* attribute value of the modified *Node* must be updated to the *CurrentConfigurationVersion*.

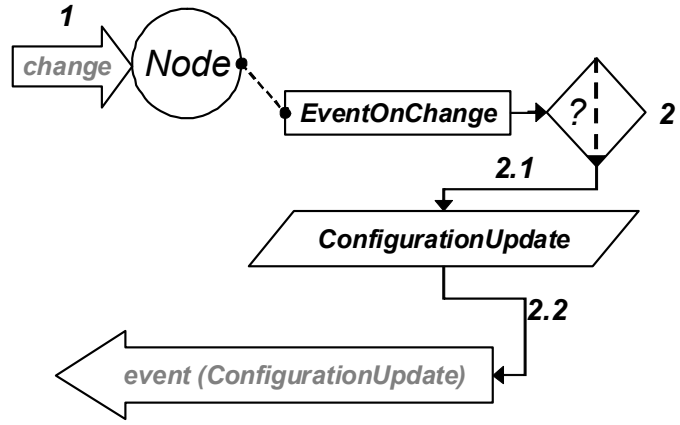

Figure 3: sequence form the *EventOnChange* attribute perspective.

The Figure 3 shows the sequence of operations in case the *EventOnChange* attribute only is supported by the *Node*. Internal steps as the *Node* value changes are the following:

1. A change occurs to the *Node*, due to an action execution or some other event out of the UPnP protocol scope.
2. If the *Node* supports the *EventOnChange* attribute and its value is 1:
   2.1. The *ConfigurationUpdate* must be updated as specified in section 2.4.1 (using the *CurrentConfigurationVersion* and the time stamp).
   2.2. The event corresponding to the *ConfigurationUpdate* state variable must be sent to the subscribed CPs.

3. If the *Node* supports the *EventOnChange* attribute and its value is 0, the *ConfigurationUpdate* must not be updated and therefore no event must be sent.

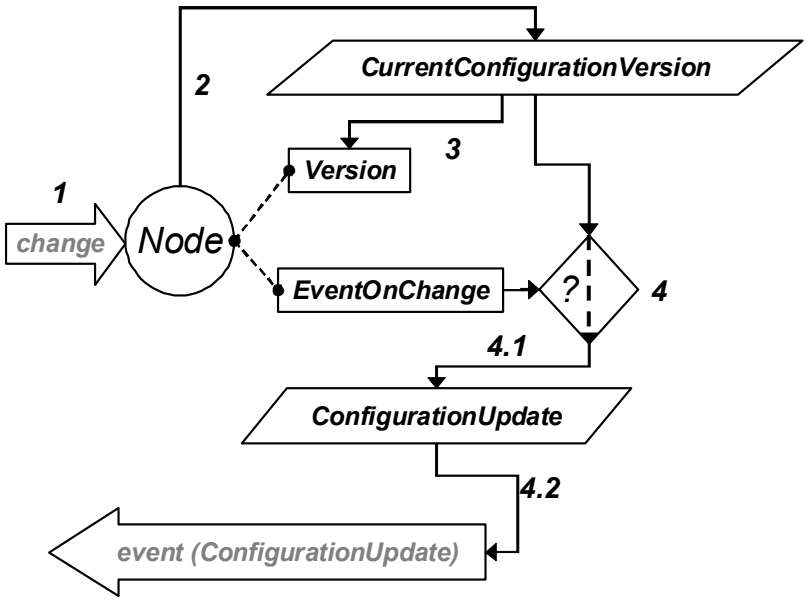

Figure 4: sequence when both *Version* and *EventOnChange* attributes are supported.

The Figure 4 shows the sequence of operations in case the both the *EventOnChange* and *Version* attributes are supported by the *Node*. Internal steps as the *Node* value changes are the following:

1. A change occurs to the *Node*, due to an action execution or some other event out of the UPnP protocol scope.
2. The *CurrentConfigurationVersion* must be updated (increased).
3. The *Version* attribute value of the modified *Node* must be updated to the *CurrentConfigurationVersion*.
4. If the *EventOnChange* attribute value is 1:
   - 4.1. The *ConfigurationUpdate* must be updated as specified in section 2.4.1 (using the *CurrentConfigurationVersion* and the time stamp).
   - 4.2. The event corresponding to the *ConfigurationUpdate* state variable must be sent to the subscribed CPs.
5. If the *EventOnChange* attribute value is 0, the *ConfigurationUpdate* must not be updated and therefore no event must be sent.

Steps numbered 3 and 4 are not a sequence and can be internally executed in parallel, depending on implementation choices.

2.5. Eventing and Moderation

Table 2-4: Event Moderation

Variable Name	Evented	Moderated Event	Max Event Rate[1]	Logical Combination	Min Delta per Event[2]
ConfigurationUpdate	*Yes*	*Yes*	0.2 seconds		
CurrentConfigurationVersion	*No*	*No*			
SupportedDataModelsUpdate	*Yes*	*Yes*	1.0 second		
SupportedParametersUpdate	*Yes*	*Yes*	1.0 second		
AttributeValuesUpdate	*Yes*	*Yes*	1.0 second		
InconsistentStatus	*Yes*	*Yes*	1.0 second		
Non-standard state variables implemented by an UPnP vendor go here.	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*

[1] Determined by N, where Rate = (Event)/(N secs).
[2] (N) # (allowedValueRange Step).

2.5.1. Event Model

This service definition is compliant with the UPnP Device Architecture version 1.0. [UDA].

2.6. Actions

There are three categories of actions defined in this service.

The first one is the "**data models discovery"** set of actions including the *GetSupportedDataModels()* and the *GetSupportedParameters()* actions. Using them properly, the CPs can discover the list of all supported *Parameters* of the *Parent Device* and where they come from (in the case of data models defined in other standardization organizations or other UPnP Working Committees).

Once the control point has the knowledge of the list of supported *Parameters*, it can use the "**status reading"** set of actions to discover the current configuration state of the particular *Parent Device*. This set includes the actions *GetInstances()*, *GetValues()*, *GetSelectedValues()*, *GetAttributes()* and *GetInconsistentStatus()*. Also *GetConfigurationUpdate()*, *GetSupportedDataModelsUpdateID()*, *GetSupportedParametersUpdateID()* and *GetAttributeValuesUpdateID()*.

The third category is the "**configuration"** set of actions used to change the current configuration state of the *Parent Device*. This set includes the actions *SetValues()*, *CreateInstance()*, *DeleteInstance()* and *SetAttributes()*.

The **configuration** actions could fail because of race conditions whenever the control point is trying to change a *Parameter* or an instance concurrently used by other entities (e.g. another control point or some other external interface), or because the targeted resource is temporarily unavailable for some reasons. In these situations it is up to the *Parent Device* implementation to resolve the concurrent access to *Parameters* and therefore the *Parent Device* MAY momentary deny the **configuration** action returning a fault code indicating this specific condition. In this situation, the control point SHOULD NOT interpret the fault code as indicating that it can not perform such action but rather as a suggestion to retry the same action later, when the conflict will disappear or the resource is available.

Immediately following this table is detailed information about these actions, including short descriptions of the actions, the effects of the actions on state variables, and error codes defined by the actions.

Table 2-5: Actions

Name	Device R/O[1]	Control Point R/O
GetSupportedDataModels()	*R*	*R*
GetSupportedParameters()	*R*	*R*
GetInstances()	*R*	*R*
GetValues()	*R*	*R*
GetAttributes()	*R*	*R*
GetConfigurationUpdate()	*R*	*O*
GetCurrentConfigurationVersion()	*R*	*O*
GetSupportedDataModelsUpdate()	*R*	*O*
GetSupportedParametersUpdate()	*R*	*O*
SetAttributes()	*O*	*O*
GetInconsistentStatus()	*O*	*O*
GetSelectedValues()	*O*	*O*
SetValues()	*O*	*O*
CreateInstance()	*O*	*O*
DeleteInstance()	*O*	*O*
GetAttributeValuesUpdate()	*O*	*O*
Non-standard actions implemented by an UPnP vendor go here.	X	X

[1] R = Required, O = Optional, X = Non-standard.

2.6.1. *GetSupportedDataModels()*

This action can be used by the control point to know which the supported data models of the *Parent Device* are, including the *Common Objects*. The *Parent Device* returns to the control point an XML fragment containing basic information as the attachment points of the supported data model and its URI (which includes, for example, the name of the data model and the version).

This action does not provide to the control point information concerning the implemented *Parameters* taken from the data models supported. For this purpose the control point must make use of the *GetSupportedParameters()* action using the Locations from *GetSupportedDataModels()* as *StartingNode* arguments.

It's important to note that this action basically deals with data model *Location* that can be interpreted as the common prefix for all *Parameters* imported from the data model. This works properly in case of both UPnP and vendor extensions compliant with this specification, but for data model imported from other organizations some conversion rules have been defined for the syntax and the semantic: see Appendix C: Mapping rules for Other .

The ouput argument is defined as follows:

SupportedDataModels

The list of the supported data models of the *Parent Device* as in *A_ARG_TYPE_SupportedDataModels* definition.

2.6.1.1. Arguments

Table 2-6: Arguments for *GetSupportedDataModels()*

Argument	Direction	relatedStateVariable
SupportedDataModels	*OUT*	*A_ARG_TYPE_SupportedDataModels*

2.6.1.2. Dependency on State (if any)

When the SMS is also implemented by the *Parent Device*, the installation and uninstallation of DUs may effect on the supported data model returned.

2.6.1.3. Effect on State (if any)

None

2.6.1.4. Errors

Table 2-7: Error Codes for *GetSupportedDataModels ()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.2. *GetSupportedParameters()*

Despite its name, this action deals with *StructurePaths*, called parameters to highlight the final purpose of the action (which is to inform the control point about the *Parameters* implemented by the device) rather than the terminology and the syntax of the returned strings (see section 2.3.1.1). This is the reason why in this action description the term *Parameter* is not written capitalized (i.e.: it does not strictly correspond to

the definition given for *Parameter*). The results returned to the control point MUST be a set of *StructurePaths* which are:

- Starting from the *Root Node* and ending to the *Leaf Nodes*.
- Starting from the *Root Node* and ending to an internal *Node* (not *Leaf Node*).

The *Parent Device* can support several data models as described in the *GetSupportedDataModels()* action. In each supported data models there could be mandatory *Parameters* as well as optional *Parameters*. The *Parent Device* MUST support every mandatory *Parameter* from the supported data models and MAY support some or all optional ones, therefore this action can be used to synchronize the control point and the *Parent Device* on the list of all supported *Parameters*. This means that, given a valid starting *Node* from one of the supported data models, the *Parent Device* will return to the control point the list of all possible (i.e. supported) *Parameters* descending from the given starting *Node*. The given starting *Node* in the data model is identified by a *StructurePath* from the *Root* to the *Node*. The *Parameters* listed by the *Parent Device* are *StructurePaths* from the *Root* to the *Leaf Nodes*.

As it can be noticed by the grammar rule defining *StructurePath*, the *MultiInstance Node* is always followed by the *InstanceAlias* (see 2.3.1.2). This is strictly necessary because the *StructurePath* is basically used to discover the structure of the data model and the control point must be able to syntactically recognize whether a *StructurePath* ending with the "/" is a *SingleInstance* or a *MultiInstance Node*. Summarizing, StructurePaths which end with

- `/…/<node_name>` are paths from the *Root* to a *Leaf Node*,
- `/…/<node_name>/` are paths from the *Root* to a *SingleInstance Node*,

`/…/<node_name>/#/` are paths from the *Root* to the *MultiInstance Node* (and following InstanceAlias).

The input arguments are defined as follows:

StartingNode

The *StartingNode* provides to the *Parent Device* the *Node* where to start the browsing. Its type is defined in the related state variable description. Passing to the *Parent Device* a *StartingNode* which ends to a *Leaf Node* is not considered a syntactical error and can be used in case the control point specifically wants to validate the existence of that *Leaf*.

SearchDepth

Due to the tree structure of the supported data model, the unsigned integer argument *SearchDepth* is used to determine how many *Nodes* to be traversed before to stop the search when browsing.

- *SearchDepth* = 0: means there is no limit to the depth of search. The Result must contain all *StructurePaths* from the *StartingNode* to the ending *Leaf Nodes* that are descendents of the *StartingNode* given. The search stops to the last *Leaf Nodes*.
- *SearchDepth* > 0: means that at most *SearchDepth* number of *Nodes* must be traversed starting from the *StartingNode*. The *Result* will contain only valid *StructurePaths* from the *Root Node* that are descendents of the given *StartingNode* (there is at least the *StartingNode* in). Such paths can end either with a *Leaf Node* or an internal *Node* as *SingleInstance* or *MultiInstance Node* followed by the *InstanceAlias* as it will be clarified in the following explanation of the *Result* argument.

The ouput argument is defined as follows:

Result

Unordered list of *StructurePaths* descending from the *StructurePath* given as *StartingNode*. Each path (i.e. sequence of *Nodes* in the parent-child relationship) in the returned list MUST be expressed as a valid *StructurePath* from the *Root Node* to and internal *Node* as well as a *Leaf Node* depending on the data model structure, the value of the *SearchDepth* and the *StartingNode* provided (see also the related state variable for the type description). This means a returned path may ends with the *Root*, a *Leaf*, a wildchar (following a *MultiInstance Node*) or a *SingleInstance Node*.

There is a special consideration for *SearchDepth* and *MultiInstance Nodes* in valid *StructurePaths* returned. The control point uses this action to discover the structure of the data model, therefore as it is specified in section 2.3.1.1, the *MultiInstance Node* which ends the path must always be followed by the *InstanceAlias*, regardless to the *SearchDepth*, in order to be properly recognized by the control point.

2.6.2.1. Arguments

Table 2-8: Arguments for *GetSupportedParameters()*

Argument	Direction	relatedStateVariable
StartingNode	*IN*	*A_ARG_TYPE_StructurePath*
SearchDepth	*IN*	*A_ARG_TYPE_SearchDepth*
Result	*OUT*	*A_ARG_TYPE_StructurePathList*

For example, if the data model of the *Parent Device* was the one represented in Figure 5:

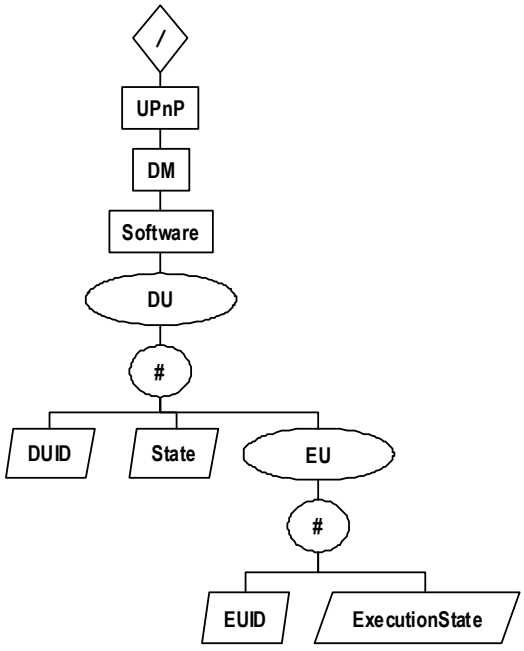

Figure 5: excerpt from SMS data model structured tree.

Using *StartingNode* = **`/UPnP/DM/Software/`** the following StructurePaths will be returned by the *Parent Device* in the *Result* argument, using different *SearchDepth* values:

SearchDepth = **0 and** *SearchDepth* **> 4** (all *StructurePaths* from *Root Node* to *Leaf Nodes*)

```
/UPnP/DM/Software/DU/#/DUID
/UPnP/DM/Software/DU/#/State
/UPnP/DM/Software/DU/#/EU/#/EUID
/UPnP/DM/Software/DU/#/EU/#/ExecutionState
```

SearchDepth = **1** (DU is the rightmost *Node* and must be recognized as a *MultiInstance Node*, therefore the *InstanceAlias* is needed)

```
/UPnP/DM/Software/DU/#/
```

SearchDepth = **2**

```
/UPnP/DM/Software/DU/#/
```

SearchDepth = **3** (EU is the rightmost *Node* and must be recognized as a *MultiInstance Node*, therefore the *InstanceAlias* is needed)

```
/UPnP/DM/Software/DU/#/DUID
/UPnP/DM/Software/DU/#/State
/UPnP/DM/Software/DU/#/EU/#
```

SearchDepth = **4**

```
/UPnP/DM/Software/DU/#/DUID
/UPnP/DM/Software/DU/#/State
/UPnP/DM/Software/DU/#/EU/#
```

2.6.2.2. Dependency on State (if any)

When the SMS is also implemented by the *Parent Device*, the installation and uninstallation of DUs may effect on the supported data model returned.

2.6.2.3. Effect on State (if any)

None

2.6.2.4. Errors

Table 2-9: Error Codes for *GetSupportedParameters()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
701	Invalid Argument Syntax	The action failed because of the wrong syntax for the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.3. GetInstances()

This action may be used by the control point to discover the list of *Instance Nodes* actually present on the *Parent Device*. Given a starting *PartialPath*, the *Parent Device* will return the list of all possible (if supported) *PartialPaths* descending from the given path.

Concerning the *PartialPaths* returned, if the path includes a *MultiInstance Node* then all *Instances* are returned, but if there are no *Instance Nodes* the search for innermost *Nodes* must stop, as it will be clearer from the examples below.

The input arguments are defined as follows:

StartingNode

The *StartingNode* is a *PartialPath* and provides to the *Parent Device* the *Node* where to start the browsing. A *StartingNode* ending with a *Leaf Node* is useless even though it is not considered an error. If the path provided to the *Parent Device* in the *StartingNode* does not exist (i.e.: its *StructurePath* does not belong to the list of supported *StructurePaths*) the *Parent Device* will respond with a fault.

SearchDepth

Since the *MultiInstance Nodes* in the supported data model can be nested, the unsigned integer argument *SearchDepth* is used to determine how many *Nodes* to be traversed before to stop the search when browsing.

- *SearchDepth* = 0: the *Result* must contain all *PartialPaths* that are descendents of the *StartingNode* given, if there exists at least an *Instance Node* in the *PartialPaths* returned. The search stops at the last *Instance Nodes*.
- *SearchDepth* > 0: the *Result* must contain all *PartialPaths* that are descendents of the *StartingNode* given, if there exists at least an *Instance Node* in the *PartialPath* returned, and such *Instance Node* is within *SearchDepth* levels of *Nodes*. Therefore the search stops after at most (but not exactly) *SearchDepth* levels of descendents where each *Node* traversed is considered a level.

The ouput argument is defined as follows:

Result

Unordered list of *InstancePaths*, descended from the *PartialPath* given in *StartingNode*. The returned list can be empty if there are no children of the given *StartingNode* traversing at least one *Instance* in the path.

2.6.3.1. Arguments

Table 2-10: Arguments for GetInstances()

Argument	Direction	relatedStateVariable
StartingNode	*IN*	*A_ARG_TYPE_PartialPath*
SearchDepth	*IN*	*A_ARG_TYPE_SearchDepth*
Result	*OUT*	*A_ARG_TYPE_InstancePathList*

The following examples will clarify better the usage of these action's arguments.

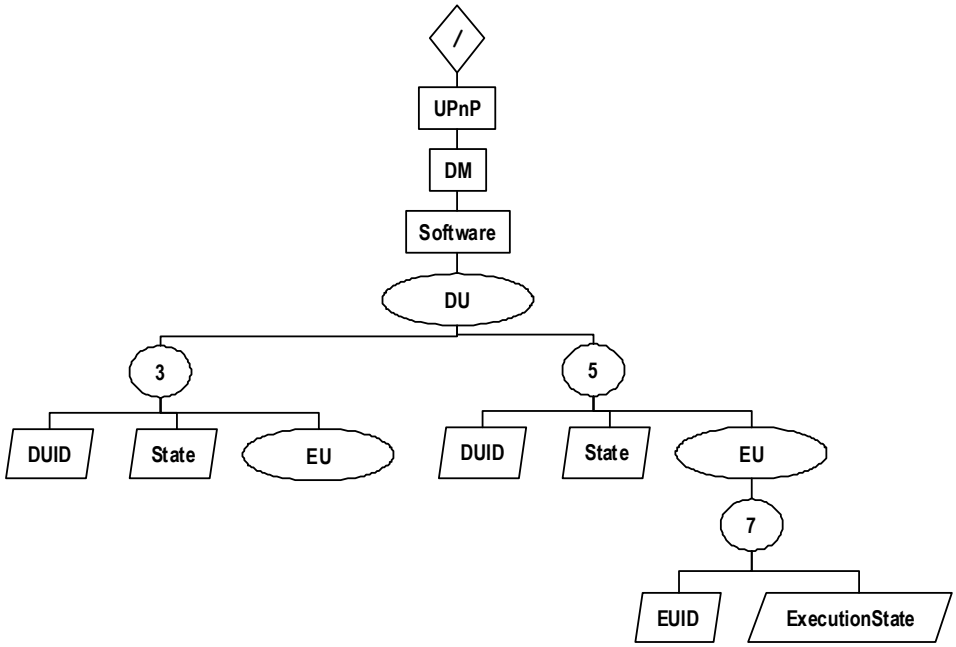

Figure 6: excerpt from SMS data model structured tree.

Using *StartingNode* = **`/UPnP/DM/Software/`** the following *InstancePaths* will be returned by the *Parent Device* in the *Result* argument, using different *SearchDepth* values:

SearchDepth = **0 and** *SearchDepth* **> 3** (all InstancePaths from *Root Node*)

```
/UPnP/DM/Software/DU/3/
/UPnP/DM/Software/DU/5/
/UPnP/DM/Software/DU/5/EU/7/
```

SearchDepth = **1**

Empty *InstancePath* list returned: there are no *Instance Nodes* within the *SearchDepth*=1 levels.

SearchDepth = **2**

```
/UPnP/DM/Software/DU/3/
/UPnP/DM/Software/DU/5/
```

SearchDepth = **3**

```
/UPnP/DM/Software/DU/3/
/UPnP/DM/Software/DU/5/
```

2.6.3.2. Dependency on State (if any)

The list of *Parameters* returned by the *Parent Device* depends on the object currently instanced.

2.6.3.3. Effect on State (if any)

None

2.6.3.4. Errors

Table 2-11: Error Codes for *GetInstances()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
701	Invalid Argument Syntax	The action failed because of the wrong syntax for the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.4. *GetValues()*

The *GetValues()* action is used to retrieve the values of one or more *Parameters* from the *Parent Device*'s data model, by passing a list of *Parameters*. The action will return a list of the required *Parameters* associated with their values. To provide more flexibility, *Parameters* could be *ParameterPaths* or *PartialPaths* as explained below.

The input argument is defined as follows:

Parameters

The control point passes to the *Parent Device* a list of *ContentPaths*. Getting the value of a *ParameterPath* in the list leads to a single parameter-value pair, whereas getting the value of other types of allowed paths can lead to a list composed of multiple Parameter-value pairs. The control point may require the same *Parameter* twice (e.g. when the both parent and child are required in the *Parameters* argument); in this situation whether to reduce the number of *Parameters* returned to avoid duplications in the response is implementation dependent.

The ouput argument is defined as follows:

ParameterValueList

The *Parent Device* must return a parameter-value pair list, in which the *Parameters* are expressed as *ParameterPaths*, containing all descendant *Parameters* of the given *ContentPath* (if any), associated with their respective values. In other words, for each *ContentPath* provided in the input argument, the entire subtree starting from such *Node* is returned. The list can be empty if none of the required input paths leads to a parameter with a value.

2.6.4.1. Arguments

Table 2-12: Arguments for *GetValues()*

Argument	Direction	relatedStateVariable
Parameters	*IN*	*A_ARG_TYPE_ContentPathList*
ParameterValueList	*OUT*	*A_ARG_TYPE_ParameterValueList*

For example, given the following *GetValues()* action *Parameters* input argument:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ContentPathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<ContentPath>/UPnP/DM/DeviceInfo/</ContentPath>
<ContentPath>/UPnP/DM/Monitoring/</ContentPath>
</cms:ContentPathList>
```

The *GetValues()* action response argument could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ParameterValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<Parameter>
<ParameterPath>/UPnP/DM/DeviceInfo/FriendlyName</ParameterPath>
<Value>The First Manageable Device</Value>
</Parameter>
<Parameter>
<ParameterPath>/UPnP/DM/DeviceInfo/ProvisioningCode</ParameterPath>
<Value>UPnP enabled custom code</Value>
</Parameter>
…
<Parameter>
<ParameterPath>/UPnP/DM/DeviceInfo/PhysicalDevice/HardwareVersion</Param
eterPath>
<Value>3.5</Value>
…
<Parameter>
<ParameterPath>/UPnP/DM/DeviceInfo/Monitoring/OperatingSystem/CPUUsage</
ParameterPath>
<Value>23</Value>
…
</cms:ParameterValueList>
```

2.6.4.2. Dependency on State (if any)

The list of *Parameters* returned by the *Parent Device* depends on the objects currently instantiated, if the *ParameterValueList* contain *Instance Nodes*.

2.6.4.3. Effect on State (if any)

None.

2.6.4.4. Errors

Table 2-13: Error Codes for *GetValues*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.

errorCode	errorDescription	Description
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
702	Invalid XML Argument	The action failed because of the wrong XML format in the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.5. GetSelectedValues()

The GetSelectedValues() optional action is used to retrieve the values of one or more *Parameters* from the *Parent Device* data model, by passing to the *Parent Device* a filter, in order to provide to allow the control point to only retrieve values in which it has a specific interest. The *Parent Device* will return the list of the queried *Parameters* along with with their associated values.

The input arguments are defined as follows:

StartingNode

The StartingNode is a *StructurePath* and may be used by the control point to narrow the possible responses in ParameterValueList to a specific subset of the data model; in this scenario, the device MUST return only *Parameter Paths* descending from the StartingNode.

Filter

The control point passes to the *Parent Device* a Filter argument as defined in the related state variable description. Only *Parameters* which satisfy the filter conditions will be returned.

The ouput argument is defined as follows:

ParameterValueList

For each parameter satisfying the given input filter, the *Parent Device* must return a parameter-value pair list. The list is unordered and includes only *Parameters* descended from the *StructurePath* given in StartingNode. The returned list can be empty if there are no descendents from the given StartingNode for the response.

2.6.5.1. Arguments

Table 2-14: Arguments for GetSelectedValues()

Argument	Direction	relatedStateVariable
StartingNode	IN	A_ARG_TYPE_StructurePath
Filter	IN	A_ARG_TYPE_Filter
ParameterValueList	OUT	A_ARG_TYPE_ParameterValueList

Example

Given the following example status in the *Parent Device*:

```
/UPnP/DM/Software/DU/7/DUID = 21
/UPnP/DM/Software/DU/7/State = "Installed"
/UPnP/DM/Software/DU/7/EU/2/EUID = 2105
/UPnP/DM/Software/DU/7/EU/2/ExecutionState = "Inactive"
/UPnP/DM/Software/DU/12/DUID = 23
/UPnP/DM/Software/DU/12/State = "Installed"
/UPnP/DM/Software/DU/12/EU/7/EUID = 2372
/UPnP/DM/Software/DU/12/EU/7/ExecutionState = "Active"
```

If the control point needs to know all information of the EUs contained by the DU identified by 23, for example, it uses the following *StructurePath* as *StartingNode* value:

```
/UPnP/DM/Software/DU/#/EU/#/
```

And the following filter:

```
/UPnP/DM/Software/DU/#/DUID = 23
```

The *ParameterValueList* in the action response will contain the following *Parameters* descending from the *StartingNode*:

```
/UPnP/DM/Software/DU/12/EU/7/EUID = 2372
/UPnP/DM/Software/DU/12/EU/7/ExecutionState = "Active"
```

The following *Parameters*:

```
/UPnP/DM/Software/DU/12/DUID = 23
/UPnP/DM/Software/DU/12/State = "Installed"
```

will not be included in the response because `/UPnP/DM/Software/DU/12/DUID` is not descended from the *StartingNode* given: `/UPnP/DM/Software/DU/#/EU/#/`

2.6.5.2. Dependency on State (if any)

The list of *Parameters* returned by the *Parent Device* depends on the objects currently instantiated if the *ParameterValueList* contain *Instance Nodes*.

2.6.5.3. Effect on State (if any)

None.

2.6.5.4. Errors

Table 2-15: Error Codes for *GetSelectedValues()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.

errorCode	errorDescription	Description
600-699	TBD	See UPnP Device Architecture section on Control.
701	Invalid Argument Syntax	The action failed because of the wrong syntax for the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device*.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.6. *SetValues()*

The *SetValues()* optional action is used to modify the state of the *Parent Device* by changing the value of one or more *Parameters* in the *Parent Device* configuration. Each action is an independent transaction, and it MUST be possible to change more *Parameters* values at once through using one *SetValues()* action.

There is a single response (either the *SetValuesResponse()* or the fault) to each *SetValues()* action, even when the action targets multiple *Parameters*. This means that, in case of success, all the changes must be saved by the *Parent Device* (commit) atomically in an all-or-nothing fashion. Otherwise, in case of failure to set one of the *Parameters* within the action, none of the required changes must be applied and the status of the *Parent Device* must return the same as before the *SetValues()* action was invoked.

If the *Parameter* is set more than once in the ParameterValueList argument, itis implementationspecific which value will be used. The *Parent Device* implementation MAY either accept multiple changes to the same *Parameter* in the same *SetValues()* action or to reject it with afault.

The input argument is defined as follows:

ParameterValueList

The control point passes to the *Parent Device* a parameter-value pair list, where the parameter names are expressed as *ParameterPaths*.

The ouput argument is defined as follows:

Status

Indicates whether the changes have been committed and applied or only committed. Depending on its internal capabilities (i.e., how the *Parent Device* manages and persistently saves configuration *Parameters*), the *Parent Device* informs the control point concerning its behavior after this *SetValues()* action terminates:

- *Status* = *ChangesCommitted* → means that changes are not yet applied: the *Parent Device* has stored new values somewhere but it is still using the old ones for the current running status. For example, for some device/service implementations the underlying operating system could need to autonomously reboot (i.e. the CMS will disappear and reappear again in the network) after the action invocation before to apply the changes. The *Parent Device* will anyway return the new values to CPs for subsequent reading action as *GetValues()* or *GetInstances()* after this *SetValues()* invocation.

- *Status* = *ChangesApplied* → means that changes have been applied and, for example, nothing else is needed by the *Parent Device* (e.g. the operating underlying system does not need to reboot). It is strongly recommended for *Parent Device* implementations to prefer this behavior rather than to delay the application of changes and use the *ChangesCommitted*.

2.6.6.1. Arguments

Table 2-16: Arguments for *SetValues()*

Argument	Direction	relatedStateVariable
ParameterValueList	*IN*	*A_ARG_TYPE_ParameterValueList*
Status	*OUT*	*A_ARG_TYPE_ChangeStatus*

2.6.6.2. Dependency on State (if any)

The list of *Parameters* to be set depends on the supported *Parameters* and on the *Instance Nodes* currently instanced.

The resulting *Status* value and the action behavior MAY be affected by the *BMS::SequenceMode* state variable value. The *BMS::SequenceMode* is a hint the *Parent Device* MAY consider to decide whether it should commit changes whether to apply them directly as it normally does. This could be useful for configuration changes that may have side effects, e.g., the change of the IP address of the *Parent Device*. Whatever the decision to commit or apply directly the changes is, the control point will be informed using the *Status* output argument value.

2.6.6.3. Effect on State (if any)

The success of the action results in the change of *Parent Device* configuration state. The change may affect targeted *Parameters* and may alsohave side-effects. All the *Parent Device* state changes may result in an increment of *CurrentConfigurationVersion* and in a *ConfigurationUpdate* change for *Parameters* (*Leaf* and *MultiInstance Nodes*) which support the *Version* and the *EventOnChange* attributes. The change of *ConfigurationUpdate* may therefore follows in an event notified to service subscribers. Refer to the specific sections and section 2.4.22 for further details.

Failures do not result in any notification. A failure results only in an error message to the requestor.

2.6.6.4. Errors

Table 2-17: Error Codes for *SetValues()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
702	Invalid XML Argument	The action failed because of the wrong XML format in the argument.

errorCode	errorDescription	Description
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
704	Invalid Value Type	The *Parameter* value has the wrong type.
705	Invalid Value	The *Parameter* value is invalid or out of range.
706	Read Only Violation	The *Parameter* is read only and cannot be set, created or deleted.
707	Multiple Set	The same *Parameter* is set more than once in the same action.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device* .
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.7. *CreateInstance()*

The *CreateInstance()* optional action is used to modify the status of the *Parent Device* by addingexactly one new *Instance Node* to a *MultiInstance Node* into the *Parent Device* configuration. The new instance is created by passing to the *Parent Device* the *PartialPath* from the *Root* to the *MultiInstance Node* (refer to the *MultiInstance* grammar rules). The *Parent Device* will return the same *PartialPath* extended with the *Instance Node* identifier (refer to the *Instance* grammar rule) that it created.

Using the *ChildrenInitialization* argument, the control point can also provide initializing values for some or all of the *Leaf Nodes* contained within the *Instance Node* to be created.

If the same *ParameterInitializationPath* is included more than once in the *ChildrenInitialization*, resulting on a multiple initialization values for the same *Parameters*, it is implementation specific which value will be used. The *Parent Device* implementation MAY accept such multiple initialization values of the same *Parameter* in the same *CreateInstance()* action, reject the action with a fault.

The input arguments are defined as follows:

MultiInstanceName

The *MultiInstanceName* argument contains the *MultiInstancePath* to identify where the *Instance Node* must be created.

ChildrenInitialization

The *ChildrenInitialization* is an XML fragment which specifies a list of name-value pairs where the names are *ParameterInitializationPaths* from *Node* of the given *MultiInstance Node* to the *Leaf* to be initialized, traversing zero or more *SingleInstance Nodes* (if the child *Node* to be initialized is nested within *SingleInstance Nodes*). The *Nodes* specified in the *ChildrenInitialization* list are optional (i.e. the list of initializing *Nodes* can be empty) and a partial subset of children is also permitted. The values are used to initialize, with the same *CreateInstance()* action, the *Nodes* contained in the *Instance* to be created. If the *Parent Device* provides the support for unique keys (see: 2.3.3), the *ChildrenInitialization* MUST be used to initialize all the *Leaf Nodes* that are part of the unique key.

The ouput arguments are defined as follows:

InstanceIdentifier

The *InstanceIdentifier* is an *InstancePath* from the *Root Node* to the *Instance Node* already created.

For example, if the control point wants to create a new *Instance Node* of a hypothetical User table, it must call the *CreateInstance()* action using "`/User/`" in the MultiInstanceName (to specify the *MultiInstancePath*). Supposing the *Parent Device* will create *Instance* number 27, it will respond to the control point the *InstancePath* "`/User/27/`" as output.

Status

See the related state variable for the type description. Depending on its internal capabilities (i.e.: how the *Parent Device* manages and persistently saves *Instance Nodes*), the *Parent Device* informs the control point concerning its behavior after this *CreateInstance()* action terminates:

- *Status* = *ChangesCommitted* → means that changes are not yet applied: the *Parent Device* have stored the new *Instance Node* somewhere but it still using the old *Instance Nodes* for the current running status. For example, for some device/service implementations the underlying operating system could need to autonomously reboot (i.e. the CMS will disappear and reappear again in the network) after the action invocation before to create the new *Instance Node* and to apply initialization values for specified children *Nodes*. The *Parent Device* will anyway return the new values to CPs for subsequent reading action as *GetInstances()* or *GetValues()* after this *CreateInstance()* invocation.
- *Status* = *ChangesApplied* → means that changes have been applied (the new *Instance Node* is created and initialization values for specified children *Nodes* have been applied) and, for example, nothing else is needed by the *Parent Device* (e.g. the operating underlying system does not need to reboot). It is strongly recommended for *Parent Device* implementations to prefer this behavior rather than to delay the application of changes and use the *ChangesCommitted*.

2.6.7.1. Arguments

Table 2-18: Arguments for *CreateInstance()*

Argument	Direction	relatedStateVariable
MultiInstanceName	*IN*	*A_ARG_TYPE_MultiInstancePath*
ChildrenInitialization	*IN*	*A_ARG_TYPE_ParameterInitialValueList*
InstanceIdentifier	*OUT*	*A_ARG_TYPE_InstancePath*
Status	*OUT*	*A_ARG_TYPE_ChangeStatus*

2.6.7.2. Dependency on State (if any)

The list of instantiable *MultiInstance Nodes* depends on the supported *parameters*.

The resulting *Status* value and the action behavior may be affected by the *BMS::SequenceMode* state variable value. The *BMS::SequenceMode* is a hint the *Parent Device* may consider to decide whether it should commit changes whether to apply them directly as it normally does. This could be useful for configuration changes that may have side effects, e.g., the change of the IP address of the *Parent Device*. Whatever the decision to commit or apply directly the changes is, the control point will be informed using the *Status* output argument value.

2.6.7.3. Effect on State (if any)

The success of the action results in the effective change of *Parent Device* configuration state. The change may affect targeted *Parameters* and may also have side-effects. All the *Parent Device* configuration state changes may result in an increment of *CurrentConfigurationVersion* and in a *ConfigurationUpdate* change for *parameters* (*Leaf* and *MultiInstance Nodes*) which support the *Version* and the *EventOnChange* attributes. The change of *ConfigurationUpdate* may therefore follows in an event notified to service subscribers. Refer to the specific sections and section 2.4.22 for further details.

Failures do not result in any notification. A failure results only in an error message to the requestor.

2.6.7.4. Errors

Table 2-19: Error Codes for *CreateInstance()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
702	Invalid XML Argument	The action failed because of the wrong XML format in the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
704	Invalid Value Type	The *Parameter* value has the wrong type.
705	Invalid Value	The *Parameter* value is invalid or out of range.
706	Read Only Violation	The *Parameter* is read only and cannot be set, created or deleted.
707	Multiple Set	The same *Parameter* is set more than once in the same action.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device* .
709	Resources Exceeded	The instance cannot be created due to lack of internal resources.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.8. *DeleteInstance()*

The *DeleteInstance()* optional action is used to delete a exactly one *Instance Node* and all its content from the *Parent Device* configuration.

The input argument is defined as follows:

InstanceIdentifier

The control point passes to the *Parent Device* an *Instance Node* identifier, expressed as an *InstancePath* from the *Root* to the *Instance Node* to be deleted.

If the *Instance Node* contains some *Nodes* that cannot be deleted, for example a critical *Parameter* for the run-time behavior of the *Parent Device* or a nested *MultiInstance Node* that must be explicitly deleted first, then the appropriate error will be returned and the action fails.

For example, to delete the *Instance* number 27 of the Network *MultiInstance Node*, the control point must call the *DeleteInstance()* action using

`/UPnP/DM/Configuration/Network/IPInterface/27/`

as the *InstanceIdentifier* argument.

If the *Parent Device* supports unique keys, the same *Instance* could also be addressed and deleted usingits unique key. For example, if the following parameter is instanced in the data model:

Value of

`/UPnP/DM/Configuration/Network/IPInterface/27/SystemName`

is `"AdvertisementInterface"`

This means that *Instance* number 27 contains a *Leaf* named `SystemName` whose value is "`AdvertisementInterface`". If the *Parent Device* support unique keys, and if and only if the `SystemName` is defined in the data model as the unique key, the control point MAY also use the following syntax to address and consequently delete the same *Instance*:

```
/UPnP/DM/Configuration/Network/IPInterface/{SystemName="AdvertisementInt
erface"}/
```

Instead of

`/UPnP/DM/Configuration/Network/IPInterface/27/`

The ouput argument is defined as follows:

Status

Depending on its internal capabilities (i.e.: how the *Parent Device* manages and persistently saves *Instance Nodes*), the *Parent Device* informs the control point concerning its behavior after this *DeleteInstance()* action terminates:

- *Status* = *ChangesCommitted* → means that changes are not yet applied: the *Parent Device* have removed the existing *Instance Node* from somewhere (e.g. the persistent memory) but it still using the old *Instance Nodes* for the current running status. For example, for some device/service implementations the underlying operating system could need to autonomously reboot (i.e. the CMS will disappear and reappear again in the network) after the action invocation before to delete the existing *Instance Node*. The *Parent Device* will anyway return the new values to CPs for subsequent reading action as *GetInstances()* or *GetValues()* after this *DeleteInstance()* invocation.

- *Status* = *ChangesApplied* → means that changes have been applied (the existing *Instance Node* is deleted) and, for example, nothing else is needed by the *Parent Device* (e.g. the operating underlying system does not need to reboot). It is strongly recommended for *Parent Device* implementations to prefer this behavior rather than to delay the application of changes and use the *ChangesCommitted*.

2.6.8.1. Arguments

Table 2-20: Arguments for DeleteInstance()

Argument	Direction	relatedStateVariable
InstanceIdentifier	*IN*	*A_ARG_TYPE_InstancePath*
Status	*OUT*	*A_ARG_TYPE_ChangeStatus*

2.6.8.2. Dependency on State (if any)

The *Instance Nodes* that can be deleted depends on the *Instance Nodes* currently instanced.

The resulting *Status* value and the action behavior MAY be affected by the *BMS::SequenceMode* state variable value. The *BMS::SequenceMode* is a hint the *Parent Device* MAY consider to decide whether it should commit changes whether to apply them directly as it normally does. This could be useful for configuration changes that may have side effects, e.g., the change of the IP address of the *Parent Device*. Whatever the decision to commit or apply directly the changes is, the control point will be informed using the *Status* output argument value.

2.6.8.3. Effect on State (if any)

The success of the action results in the change of *Parent Device* configuration state. The change will affect targeted *Parameters* and MAY also have side-effects on other *Parameters* as well. All the *Parent Device* configuration state changes MAY result in *CurrentConfigurationVersion* incrementing and in a *ConfigurationUpdate* change for *Parameters* (*Leaf* and *MultiInstance Nodes*) which support the *Version* and the *EventOnChange* attributes. The change of *ConfigurationUpdate* MAY therefore be followed by an event notified to service subscribers. Refer to the specific sections and section 2.4.22 for further details.

Failures do not result in any notification. A failure results only in an error message to the requestor.

2.6.8.4. Errors

Table 2-21: Error Codes for DeleteInstance()

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
706	Read Only Violation	The *Parameter* is read only and cannot be set, created or deleted.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device*.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.9. GetAttributes()

The GetAttributes() action is used to retrieve the attribute values of *Parameters* and *MultiInstance Nodes* from the *Parent Device* data model, by passing to the *Parent Device* a list of *ParameterPaths*, *MultiInstancePaths* or *InstancePaths* (see section 2.3.2 for further details on attributes).

The *Parent Device* will return a list of *Parameters* and *MultiInstance Node* with their associated attribute values.

As stated in section 2.3.2, not all *Nodes* have attributes. and therefore can be included in the response. The attributes are returned for the list of input parameters only.

The input argument is defined as follows:

Parameters

The control point passes to the *Parent Device* a list of:

- *ParameterPaths*,
- *MultiInstancePaths* or
- *InstancePaths*.

that could be mixed (see the related state variable for the type description).

The control point MAY require the same *Parameter* twice: it's up to the device implementation to define whether to reduce the number of *Parameters* returned to avoid duplications in the response. The list can be empty if none of the required *paths* leads to a *Node* which is supported by the data model and has at least one attribute.

The ouput argument is defined as follows:

NodeAttributeValueList

The *Parent Device* MUST return an XML string, containing exactly the same list of paths that were provided as arguments with the list of their associated attributes values. If a given *path* does not have attribute values the device must not include such a *path* in the returned list.

2.6.9.1. Arguments

Table 2-22: Arguments for GetAttributes()

Argument	Direction	relatedStateVariable
Parameters	*IN*	*A_ARG_TYPE_NodeAttributePathList*
NodeAttributeValueList	*OUT*	*A_ARG_TYPE_NodeAttributeValueList*

Example

For example, given the following GetAttributes() action input argument:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:NodeAttributePathList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<NodeAttributePath>/UPnP/DM/DeviceInfo/FriendlyName</NodeAttributePath>
<NodeAttributePath>/UPnP/DM/DeviceInfo/PhysicalDevice/NetworkInterface/<
/NodeAttributePath>
<NodeAttributePath>/UPnP/DM/Configuration/Network/IPInterface/3/</NodeAt
tributePath>
</cms:NodeAttributePathList>
```

The *GetAttributes()* action response argument could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:NodeAttributeValueList xmlns:cms="urn:schemas-upnp-org:dm:cms"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
http://www.upnp.org/schemas/dm/cms.xsd">
<Node>
<NodeAttributePath>/UPnP/DM/DeviceInfo/FriendlyName</NodeAttributePath>
<Type>string(64)</Type>
<Access>readWrite</Access>
<EventOnChange>0</EventOnChange>
</Node>

<Node>
<NodeAttributePath>/UPnP/DM/DeviceInfo/PhysicalDevice/NetworkInterface/<
/NodeAttributePath>
<Access>readOnly</Access>
<EventOnChange>1</EventOnChange>
</Node>

<Node>
<NodeAttributePath>/UPnP/DM/Configuration/Network/Interface/3/</NodeAttr
ibutePath>
<Access>readOnly</Access>
<EventOnChange>1</EventOnChange>
</Node>
</cms:NodeAttributeValueList>
```

2.6.9.2. Dependency on State (if any)

The list of *Parameter* attributes returned by the *Parent Device* depends on the supported data model and on the *Instance Nodes* currently instantiated.

2.6.9.3. Effect on State (if any)

None.

2.6.9.4. Errors

Table 2-23: Error Codes for *GetAttributes()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.

errorCode	errorDescription	Description
600-699	TBD	See UPnP Device Architecture section on Control.
702	Invalid XML Argument	The action failed because of the wrong XML format in the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device*.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.10. *SetAttributes()*

The *SetAttributes()* optional action is used to set the values of ReadWrite attributes for *Parameters* and *MultiInstance Nodes* from the *Parent Device* data model, by passing to the *Parent Device* a list of *ParameterPaths* or *MultiInstancePaths* (see section 2.3.2 for further details on attributes).

There is a single response (either the *SetAttributesResponse()* or the fault) to multiple set commanded with the same *SetAttributes()* action because the response is related to the entire *SetAttributes()* action rather than to each set individually. This means that, in case of success, all the changes must be saved by the *Parent Device* (*commit*). Otherwise, in case of failure of one of the single set commanded within the same action invocation, none of the required changes must be applied and the status of the *Parent Device* must return the same as before the *SetAttributes()* action was invoked (*rollback*).

The input argument is defined as follows:

NodeAttributeValueList

The control point passes to the *Parent Device* an XML string (see the related state variable for the type description) containing a mixture of *MultiInstancePaths* or *ParameterPaths*associated with attribute values for ReadWrite attributes only (ReadOnly attributes cannot be changed, hence set, by the control point).

Paths provided to the *Parent Device* can be:

- *MultiInstancePaths* to set attribute values of intermediate *MultiInstance Nodes*,
- *ParameterPaths*, to set attribute values of *Leaf Nodes*.

As stated in section 2.3.2, only *MultiInstance Nodes* and *Parameters* (*Leaf Nodes*) have ReadWrite attributes and can be valid input arguments for the *SetAttributes()* action.

InstancePaths are also allowed in *NodeAttributeValueList* argument but the *Access* attribute associated to *InstancePaths* are ReadOnly, therefore an attempt to set its value will cause a fault code returned by the device (e.g. “Read Only Violation”).

The ouput argument is defined as follows:

Status

Depending on its internal capabilities (i.e.: how the *Parent Device* manages and persistently saves attribute values), the *Parent Device* informs the control point concerning its behavior after this *SetAttributes()* action terminates:

- *Status* = *ChangesCommitted* → means that changes are not yet applied: the *Parent Device* have stored the new attribute values somewhere but it still using the old values for the current running status. For example, for some device/service implementations the underlying operating system could need to autonomously reboot (i.e. the CMS will disappear and reappear again in the network) after the action invocation to apply the changes The *Parent Device* will anyway return the new values to CPs for subsequent reading action as *GetAttributes()* after this *SetAttributes()* invocation.
- *Status* = *ChangesApplied* → means that changes have been applied and, for example, nothing else is needed by the *Parent Device* (e.g. the operating underlying system does not need to reboot). It is strongly recommended for *Parent Device* implementations to prefer this behavior rather than to delay the application of changes and use the *ChangesCommitted*.

2.6.10.1.Arguments

Table 2-24: Arguments for *SetAttributes()*

Argument	Direction	relatedStateVariable
NodeAttributeValueList	*IN*	*A_ARG_TYPE_NodeAttributeValueList*
Status	*OUT*	*A_ARG_TYPE_ChangeStatus*

2.6.10.2.Dependency on State (if any)

The list of attributes that can be set depends on the supported data model and on the *Instance Nodes* currently instanced.

The resulting *Status* value and the action behavior MAY be affected by the *BMS::SequenceMode* state variable value. The *BMS::SequenceMode* is a hint the *Parent Device* MAY consider to decide whether it should commit changes whether to apply them directly as it normally does. This could be useful for configuration changes that may have side effects, e.g., the change of the IP address of the *Parent Device*. Whatever the decision to commit or apply directly the changes is, the control point will be informed using the *Status* output argument value.

2.6.10.3.Effect on State (if any)

The success of the action results in the effective change of *Parent Device* data. The change may affect targeted *Parameters* and may have side-effects. All the *Parent Device* data changes may result in an increment of *CurrentConfigurationVersion* and in a *ConfigurationUpdate* change for *Parameters* (*Leaf* and *MultiInstance Nodes*) which support the *Version* and the *EventOnChange* attributes. The change of *ConfigurationUpdate* may therefore follows in an event notified to service subscribers. Refer to the specific sections and section 2.4.22 for further details.

Failures do not result in any notification. A failure results only in an error message to the requestor.

2.6.10.4.Errors

Table 2-25: Error Codes for *SetAttributes()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
702	Invalid XML Argument	The action failed because of the wrong XML format in the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
704	Invalid Value Type	The *Parameter* value has the wrong type.
705	Invalid Value	The *Parameter* value is invalid or out of range.
706	Read Only Violation	The *Parameter* is read only and cannot be set, created or deleted.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device*.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6.11.*GetInconsistentStatus()*

The *GetInconsistentStatus()* optional action can be used by CPs that have not subscribed to receive changes to the *InconsistentStatus* state variable in order to check whether the status of the *Parent Device* is consistent. This action MUST be implemented if the *InconsistenStatus* optional state variable is implemented.

The ouput argument is defined as follows:

StateVariableValue

The *Parent Device* returns to the control point the value of the *InconsistentStatus* state variable.

2.6.11.1.Arguments

Table 2-26: Arguments for *GetInconsistentStatus()*

Argument	Direction	relatedStateVariable
StateVariableValue	*OUT*	*InconsistentStatus*

2.6.11.2.Dependency on State (if any)

The value of the returned status depends on the value of the *InconsistentStatus* state variable.

2.6.11.3.Effect on State (if any)

None

2.6.11.4.Errors

Table 2-27: Error Codes for *GetInconsistentStatus()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.12.*GetConfigurationUpdate()*

The *GetConfigurationUpdate()* action can be used by CPs that have not subscribed to receive changes to the *ConfigurationUpdate* state variable in order to to read the value of the state variable.

2.6.12.1.Arguments

Table 2-28: Arguments for *GetConfigurationUpdate()*

Argument	Direction	relatedStateVariable
StateVariableValue	*OUT*	*ConfigurationUpdate*

2.6.12.2.Dependency on State (if any)

The value of the returned status depends on the value of the *ConfigurationUpdate* state variable.

2.6.12.3.Effect on State (if any)

None

2.6.12.4.Errors

Table 2-29: Error Codes for *GetConfigurationUpdate()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.13.*GetCurrentConfigurationVersion()*

The *GetCurrentConfigurationVersion()* action can be used by CPs that have not subscribed to receive changes to the *CurrentCOnfigurationVersion* state variable in order to read the value of the state variable.

2.6.13.1.Arguments

Table 2-30: Arguments for *GetCurrentConfigurationVersion()*

Argument	Direction	relatedStateVariable
StateVariableValue	*OUT*	*CurrentConfigurationVersion*

2.6.13.2.Dependency on State (if any)

The value of the returned status depends on the value of the *CurrentConfigurationVersion* state variable.

2.6.13.3.Effect on State (if any)

None

2.6.13.4.Errors

Table 2-31: Error Codes for *GetCurrentConfigurationVersion()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.14.*GetSupportedDataModelsUpdate()*

The *GetSupportedDataModelsUpdate()* action can be used by CPs that have not subscribed to receive changes to the *SupportedDataModelsUpdate* state variable in order to read the value of the state variable.

2.6.14.1.Arguments

Table 2-32: Arguments for *GetSupportedDataModelsUpdate()*

Argument	Direction	relatedStateVariable
StateVariableValue	*OUT*	*SupportedDataModelsUpdate*

2.6.14.2.Dependency on State (if any)

The value of the returned status depends on the value of the *SupportedDataModelsUpdate* state variable.

2.6.14.3.Effect on State (if any)

None

2.6.14.4.Errors

Table 2-33: Error Codes for *GetSupportedDataModelsUpdate()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.15.*GetSupportedParametersUpdate()*

The *GetSupportedParametersUpdate()* action can be used by CPs that have not subscribed to the *SupportedParametersUpdate* events to read the value of the state variable.

2.6.15.1.Arguments

Table 2-34: Arguments for *GetSupportedParametersUpdate()*

Argument	Direction	relatedStateVariable
StateVariableValue	*OUT*	*SupportedParametersUpdate*

2.6.15.2.Dependency on State (if any)

The value of the returned status depends on the value of the *SupportedParametersUpdate* state variable.

2.6.15.3.Effect on State (if any)

None

2.6.15.4.Errors

Table 2-35: Error Codes for *GetSupportedParametersUpdate()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.16.*GetAttributeValuesUpdate()*

The *GetAttributeValuesUpdate()* optional action can be used by CPs that have not subscribed to the *AttributeValuesUpdate* events to read the value of the state variable. This action MUST be implemented if the *AttributeValuesUpdate* optional state variable is implemented.

2.6.16.1.Arguments

Table 2-36: Arguments for *GetAttributeValuesUpdate()*

Argument	Direction	relatedStateVariable
StateVariableValue	*OUT*	*AttributeValuesUpdate*

2.6.16.2.Dependency on State (if any)

The value of the returned status depends on the value of the *AttributeValuesUpdate* state variable.

2.6.16.3.Effect on State (if any)

None

2.6.16.4.Errors

Table 2-37: Error Codes for *GetAttributeValuesUpdate()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.6.17.Non-Standard Actions Implemented by a UPnP Vendor

To facilitate certification, non-standard actions implemented by UPnP vendors should be included in this service template. The UPnP Device Architecture [UDA] lists naming requirements for non-standard actions (see the section on Description).

2.6.18.Relationships Between Actions

Add any summary information regarding dependencies between actions. Delete this entire section if you are not adding any summary information.

2.6.19.Common Error Codes

The following table lists error codes common to actions for this service type. If an action results in multiple errors, the most specific error must be returned.

Table 2-38: Common Error Codes

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.

errorCode	errorDescription	Description
600-699	TBD	See UPnP Device Architecture section on Control.
700		Reserved for future extensions
701	Invalid Argument Syntax	The action failed because of the wrong syntax for the argument.
702	Invalid XML Argument	The action failed because of the wrong XML format in the argument.
703	No Such Name	One or more *Parameters* given to action argument do not exist in the supported/implemented data model.
704	Invalid Value Type	The *Parameter* value has the wrong type.
705	Invalid Value	The *Parameter* value is invalid or out of range.
706	Read Only Violation	The *Parameter* is read only and cannot be set, created or deleted.
707	Multiple Set	The same *Parameter* is set more than once in the same action.
708	Resource Temporarily Unavailable	The resources required for this action cannot be internally accessed due to a concurrency problem or some other temporarily problem in the *Parent Device*.
709	Resources Exceeded	The instance cannot be created due to lack of internal resources.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.7. Theory of Operation

This section walks through several scenarios to illustrate the various actions supported by the *ConfigurationManagement:1* service.

2.7.1. Discovering of the Data Model

The *GetSupportedDataModels()* and the *GetSupportedParameters()* actions allow a control point to discover the data model's structure of a *Parent Device*.

The *GetSupportedDataModels()* returns the list of all data model definitions supported by the device. Those definitions include at least the *Common Objects*, which is the definition of the minimal set of *Parameters* that are supported by all *Parent Device* instances.

The data model of a device is composed by the *Common Objects* and might be enriched using more *Parameters*. Such *Parameters* might be described in other data model definitions and grouped in a global tree structure. This tree structure is not guaranteed to be the same for each *Parent Device*, that is why the *GetSupportedDataModels()* action returns also a location path where each data model definition is incorporated.

The *GetSupportedParameters()* action allows a control point to discover which *Parameters*, in the structure of the supported data model, are currently supported by the device. The meaning (semantic) of

each *Parameter* comes from the data model definition (e.g.: OMA-DM objects, TR-106) and should be known by the control point if it needs to properly manage them.

Using the combination of *GetSupportedDataModels()* and *GetSupportedParameters()*, the control point can build an internal view of the entire data model structure supported by a *Parent Device*.

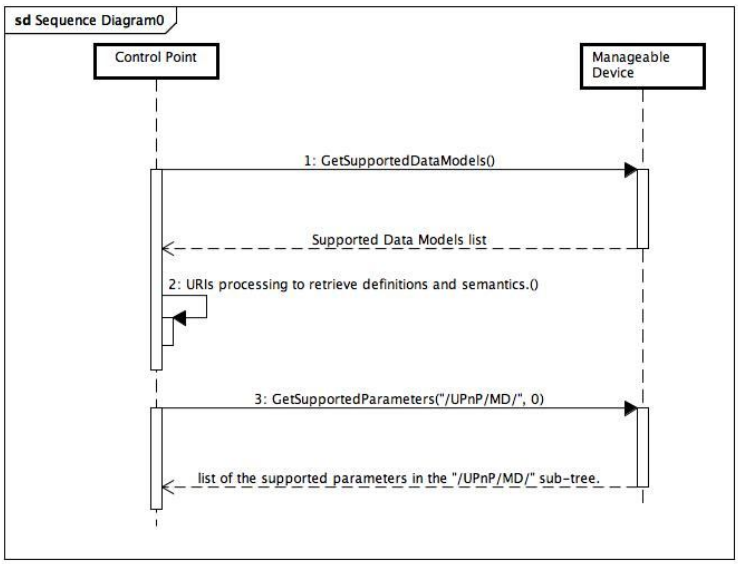

Figure 7: sequence for discovering the supported data model and parameters.

Here is a sequence of actions to achieve that goal (see Figure 7):

1. Control point calls *GetSupportedDataModels()*, and receives as the result an XML formatted list of data model definitions currently supported by the device. Control point parses the XML returned value to retrieve all the definitions' *paths* that it is able to understand. As a generic control point for MDs, it only have to understand the definition identified by the URI `urn:UPnP:ManageableDevice:1:CommonObjects:1` which is the *Common Objects*' definition. The local path associated to this data model definition is `"/UPnP/DM"`. A priori knowledge in the control point is needed to correctly interpret and manage the information about other data models.

2. Control point calls *GetSupportedParameters()* using `"/UPnP/DM"` as starting *Node* with *SearchDepth* set to 0. The control point limits the search to the sub-tree descendant of `"/UPnP/DM"`. The search depth "0" means that the control point wants to retrieve the whole sub-tree. Alternatively, if the control point is interested to retrieve all the *Parameters* supported it has to call the *GetSupportedParameters*(`"/",0`) instead.

At this stage, the control point knows the *Common Objects* structure currently supported by the *Parent Device*, i.e., it knows what optional *Parameters* are present or not.

The list of supported data model definitions and supported *Parameters* can change during the lifetime of the device. Any change results in the generation of an event that allows a control point that has subscribed to events to know when it is useful to re-discover the data model. If the control point does not use an event based logic, then it is up to the control point the decide when to re-discover the data model.

2.7.2. Management

The data model is the right place to search information concerning the configuration and the actual state of the device. A control point can use the *GetValues()* and the optional *SetValues()* and *GetSelectedValues()* to operate a trouble-shooting session. In the following example let's assume that the *SetValues()* and the *GetSelectedValues()* actions are implemented and that the device is having problem communicating with services available on the Internet. In our example the device has got only one network interface also used for UPnP management, i.e., connectivity is available and only the internet access is having trouble.

- Control point calls *GetSelectedValues(*"…/UPnP/DM/Configuration/Network/IPInterface/1/IPv4/…", ,""*)* where the first argument is the common prefix of all *Parameters* that will be returned and the second argument is an empty filter. The common prefix, here, is the *Root* of the sub-tree containing the IP configuration of the network interface.

- Control point checks the validity of the value of all returned *Parameters*. In our example everything is correct except that the value of the /UPnP/DM/Configuration/Network/IPInterface/1/IPv4/DNSServers *Parameters* is an empty string. This *Parameter* contains the list of DNS servers to query to resolve IP addresses. As the value is currently empty, the device is not able to resolve IP addresses and therefore to access properly the Internet services.

- Control point calls *SetValues(*/UPnP/DM/Configuration/Network/IPInterface/1/IPv4/DNSServers, "212.123.195.200, 212.123.195.201"*)* to update the configuration. The first argument is the *Parameter* to modify; the second argument is the new value to set.

Alternatively, if the *GetSelectedValues()* action is not implemented by the device, the control point will call the *GetValues()* action with the list of *Parameters* to retrieve as input argument value.

2.7.3. BMS Interaction

The *BMS::SetSequenceMode()* action is an optional action of the *BasicManagement:1* service (BMS). It allows a control point to indicate to the *Parent Device* that it plans to execute a sequence of actions. From the *ConfigurationManagement:1* Service (CMS) point of view, the sequence mode handled by the BMS is a hint that can be taken into account to decide not to instantly apply changes. This hint may, for instance, influence the behavior of the *SetValues()* action.

When the control point needs to configure the *Parent Device* by executing a sequence of one or more configuration actions, the *BMS::SetSequenceMode()* action can be used to inform the *Parent Device* of the beginning and the end of such configuration session. This is really useful whenever the *Parent Device* needs to do some time-consuming operations (e.g. a reboot of the underlying operating system, which may happen in some simple devices), after the control point invokes actions like, for example, *SetValues()* or *DeleteInstance()*. Refer to [BMS] for further details on *BMS::SetSequenceMode()* action and its usage.

Let's take as example a *Parent Device* targeting a Linux system. We assume that the update of the *Parameter* "/UPnP/DM/Configuration/Network/HostName" requires the reboot of the device to be applied. We also assume that the update of the *Parameter* "/UPnP/DM/Configuration/Network/IPInterface/#/IPv4/AddressingType" requires the reset of the network connection to be applied. The change of the "/UPnP/DM/DeviceInfo/FriendlyName" *Parameter* can be applied instantly. The Control Point desires not to be interrupted while executing those 3 updates one after the other. It can then use the sequence mode to reduce the probability to see the *Parent Device* disappear before it can request all the changes it is planning to apply.

- Control point calls *BMS::SetSequenceMode*("1"). The control point informs the device it is planning to execute a sequence of actions and desires not to be interrupted by side effects of the appliance of configuration changes.
- Control point calls *SetValues*("/UPnP/DM/Configuration/Network/HostName", "myNewHostName") to update the configuration. At this step the device should avoid to apply changes and therefore to reboot.
- Control point calls *SetValues*("/UPnP/DM/Configuration/Network/IPInterface/1/IPv4/AddressingType","dhcp") to update the configuration. At this step the device should avoid to apply changes and therefore to reset the network connectivity.
- Control point calls *SetValues*("/UPnP/DM/DeviceInfo/FriendlyName","myNewFriendlyName") to update configuration. At this step the device can apply changes.

Control point calls *BMS::SetSequenceMode*("0"). The control point informs the device it has completed the sequence of action call. The device can now apply all the changes not yet applied. The device will reboot as soon as possible which will cause the network connection to be reset.

2.7.4. Eventing from Changes in *Parameter* Values

The data model contains valuable information concerning the configuration of the device. Changes in the configuration may impact the behavior of the device. The eventing mechanism allows control point to be informed each time some *Parameter* values change. Let's take the example where a control point want to know each time a device changes its hostname. The information is store in the data model using the "/UPnP/DM/Configuration/Network/HostName" *Parameter*.

- Control point calls *SetAttributes*(""). The first argument is the path to the HostName *Parameter* and the value of the *EventOnChange* attribute set to 1. By doing so the control point asks the device to send an event each time the value of the HostName *Parameter* changes.
- The hostname of the device is updated by any means, e.g. the call to *SetValues()* or due to a DHCP request. The *Parent Device* sends an event to all control points that have subscribed to events. The event contains the value and the timestamp of the last change of the *CurrentConfigurationVersion* state variable.
- The control point calls *GetValues*("/UPnP/DM/Configuration/Network/HostName") to check if the value of the hostname has been changed.
- Control point calls *SetAttributes*(""). The first argument is the path to the HostName *Parameter* and the value of the *EventOnChange* attribute set to 0. By doing so the control point asks the device NOT to send an event each time the value of the HostName *Parameter* changes.
- The hostname of the device is updated by any means, e.g. the call to *SetValues()* or due to a DHCP request. The *Parent Device* does not send any event.

The eventing mechanism offered by the use of the *EventOnChange* attribute can be extended using the support of the version attribute. See next section for more details.

2.7.5. *Version* Control

Some *Nodes* of the data model support the *Version* attribute. When the related *Parameter* is updated, this attribute assumes the integer value of the *CurrentConfigurationVersion* state variable. The value of this attribute can be used as part of a filter in the *GetSelectedValues()* action call.This can be useful for a control point to compute the difference between the image of the data model it stored locally and the actual values read from the device.

The version might also be used by the control point to retrieve which are the "last" changed *Parameters* unless it is able to associate a number (the version value) to something specific (a particular configuration session). In case the control point is interested to monitor which *Parameter* change its value on a 24 hours basis, it reads the *CurrentConfigurationVersion* and save its value and, after 24 hours queries the DM using *GetSelectedValues()* asking for all *Parameters* where the *Version* value is greater that the *CurrentConfigurationVersion* previously saved. In this way it would be able to determine which are the *Parameters* whose value changed in the meantime.

In the following example, let's assume that all the *Parameters* we will deal with support the *EventOnChange* and the *Version* attribute.

- Control point subscribes to *ConfigurationManagement:1* Service events.
- The *Parent Device* sends to all subscribers the list of the evented state variables and their value. As part of this list, the *ConfigurationUpdate* state variable value contains the current configuration version.
- Control point stores locally the value of the current configuration version for later use.
- Control point calls *SetAttributes(""). The first argument is the list of paths to all the *Parameters* the control point is interested in and the value of the *EventOnChange* attribute set to 1 for all of them. By doing so the control point asks the device to send an event each time the value of one of these *Parameters* changes.
- The hostname of the device is updated by any means, e.g. the call to *SetValues()* or due to a DHCP request. The *ManageableDevie* reflects the changes by incremented by one the *CurrentConfigurationVersion* state variable and by affecting this new value to the *Version* attribute of the newly updated *Parameter*. The *Parent Device* sends an event to all control points that have subscribed to events. The event contains the value and the timestamp of the last change of the *CurrentConfigurationVersion* state variable.
- Control point detects the changes in the *CurrentConfigurationVersion* using the content of the event. It means that at least one *Parameter* that supports the *Version* attribute has been updated.

Control point calls the *GetSelectedValues()* action to retrieve all the *Parameters* that have a version higher than the one it has stored when it received the initial event after subscription. It will allow the control point to get the latest values of the *Parameters* under version control all in once.

2.7.6. *MultiInstance Nodes* Management

The *CreateInstance()* and *DeleteInstance()* actions are optional. When supported it allows control points to create and delete instances, i.e., children of *MultiInstance Nodes*. These 2 actions can only be used on *MultiInstance Nodes* with readWrite accesses. The *Common Objects* does not bring a MultiInstanceNode with readWrite accesses; so for the sake of the example, we will assume that the hypothetical `/UPnP/DM/Configuration/LocalUsersAndGroups/Users` *MultiInstance Node* exists with the readWrite accesses. Each instance corresponds to a local user defined on the device. In the following

example a control point will create a user B then delete an already existing user A. The discovery of the data model is considered as already done.

- Control point calls *CreateInstance(*`"/UPnP/DM/Configuration/LocalUsersAndGroups/Users", "Login = sshuser"`*)*; where the first argument is the *MultiInstance Node* in which to create an instance. The second argument is the list of *Parameters* and their value for the initialization.
- Control point calls *GetInstances(*`"/UPnP/DM/Configuration/LocalUsersAndGroups/Users",0,`*)*
- Control point calls *DeleteInstance(*`"/UPnP/DM/Configuration/LocalUsersAndGroups/Users/1"`*)*;

2.7.7. SMS Interaction

The Software Management Service (SMS) manages its own sub-tree in the data model. This sub-tree is often called the Software Data Model in the specification documents. The *SMS::Install()* and *SMS::Uninstall()* actions are respectively responsible of the creation and the deletion of instances in the Software Data Model. Those instances are children of the `/UPnP/DM/Software/DU` or `/UPnP/DM/Software/DU/#/EU` *MultiInstance Nodes*. *Nodes* created by the SMS are not different from any *Node* in the data model. Control points can manipulate them using the actions provided by the *ConfigurationManagement:1* Service.

2.7.8. Consistency

The *ConfigurationManagement:1* Service brings the notion of changes that are committed and changes that are applied.

3. XML Service Description

```
<?xml version="1.0"?>
<s:scpd xmlns:s="urn:schemas-upnp-org:service-1-0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:schemas-upnp-org:service-1-0 service-1-0.xsd">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>

  <actionList>

    <action>
      <name>GetSupportedDataModels</name>
      <argumentList>
        <argument>
          <name>SupportedDataModels</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_SupportedDataModels</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetSupportedParameters</name>
      <argumentList>
        <argument>
          <name>StartingNode</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_StructurePath</relatedStateVariable>
        </argument>
        <argument>
          <name>SearchDepth</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_SearchDepth</relatedStateVariable>
        </argument>
        <argument>
          <name>Result</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_StructurePathList</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetInstances</name>
      <argumentList>
        <argument>
          <name>StartingNode</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_PartialPath</relatedStateVariable>
        </argument>
        <argument>
          <name>SearchDepth</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_SearchDepth</relatedStateVariable>
        </argument>
        <argument>
          <name>Result</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_InstancePathList</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetValues</name>
      <argumentList>
        <argument>
          <name>Parameters</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ContentPathList</relatedStateVariable>
```

```
        </argument>
        <argument>
          <name>ParameterValueList</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ParameterValueList</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <Optional/>
      <name>GetSelectedValues</name>
      <argumentList>
        <argument>
          <name>StartingNode</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_StructurePath</relatedStateVariable>
        </argument>
        <argument>
          <name>Filter</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_Filter</relatedStateVariable>
        </argument>
        <argument>
          <name>ParameterValueList</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ParameterValueList</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <Optional/>
      <name>SetValues</name>
      <argumentList>
        <argument>
          <name>ParameterValueList</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ParameterValueList</relatedStateVariable>
        </argument>
        <argument>
          <name>Status</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ChangeStatus</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <Optional/>
      <name>CreateInstance</name>
      <argumentList>
        <argument>
          <name>MultiInstanceName</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_MultiInstancePath</relatedStateVariable>
        </argument>
        <argument>
          <name>ChildrenInitialization</name>
          <direction>in</direction>

<relatedStateVariable>A_ARG_TYPE_ParameterInitialValueList</relatedStateVariable>
        </argument>
        <argument>
          <name>InstanceIdentifier</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_InstancePath</relatedStateVariable>
        </argument>
        <argument>
          <name>Status</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ChangeStatus</relatedStateVariable>
        </argument>
      </argumentList>
    </action>
```

```
<action>
  <Optional/>
  <name>DeleteInstance</name>
  <argumentList>
    <argument>
      <name>InstanceIdentifier</name>
      <direction>in</direction>
      <relatedStateVariable>A_ARG_TYPE_InstancePath</relatedStateVariable>
    </argument>
    <argument>
      <name>Status</name>
      <direction>out</direction>
      <relatedStateVariable>A_ARG_TYPE_ChangeStatus</relatedStateVariable>
    </argument>
  </argumentList>
</action>

<action>
  <name>GetAttributes</name>
  <argumentList>
    <argument>
      <name>Parameters</name>
      <direction>in</direction>
      <relatedStateVariable>A_ARG_TYPE_NodeAttributePathList</relatedStateVariable>
    </argument>
    <argument>
      <name>NodeAttributeValueList</name>
      <direction>out</direction>
      <relatedStateVariable>A_ARG_TYPE_NodeAttributeValueList</relatedStateVariable>
    </argument>
  </argumentList>
</action>

<action>
  <Optional/>
  <name>SetAttributes</name>
  <argumentList>
    <argument>
      <name>NodeAttributeValueList</name>
      <direction>in</direction>
      <relatedStateVariable>A_ARG_TYPE_NodeAttributeValueList</relatedStateVariable>
    </argument>
    <argument>
      <name>Status</name>
      <direction>out</direction>
      <relatedStateVariable>A_ARG_TYPE_ChangeStatus</relatedStateVariable>
    </argument>
  </argumentList>
</action>

<action>
  <Optional/>
  <name>GetInconsistentStatus</name>
  <argumentList>
    <argument>
      <name>StateVariableValue</name>
      <direction>out</direction>
      <relatedStateVariable>InconsistentStatus</relatedStateVariable>
    </argument>
  </argumentList>
</action>

<action>
  <name>GetConfigurationUpdate</name>
  <argumentList>
    <argument>
      <name>StateVariableValue</name>
      <direction>out</direction>
      <relatedStateVariable>ConfigurationUpdate</relatedStateVariable>
    </argument>
  </argumentList>
</action>
```

```
    <action>
      <name>GetCurrentConfigurationVersion</name>
      <argumentList>
        <argument>
          <name>StateVariableValue</name>
          <direction>out</direction>
          <relatedStateVariable>CurrentConfigurationVersion</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetSupportedDataModelsUpdate</name>
      <argumentList>
        <argument>
          <name>StateVariableValue</name>
          <direction>out</direction>
          <relatedStateVariable>SupportedDataModelsUpdate</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetSupportedParametersUpdate</name>
      <argumentList>
        <argument>
          <name>StateVariableValue</name>
          <direction>out</direction>
          <relatedStateVariable>SupportedParametersUpdate</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <Optional/>
      <name>GetAttributeValuesUpdate</name>
      <argumentList>
        <argument>
          <name>StateVariableValue</name>
          <direction>out</direction>
          <relatedStateVariable>AttributeValuesUpdate</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

  </actionList>

  <serviceStateTable>

    <stateVariable sendEvents="yes"  >
      <name>ConfigurationUpdate</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>CurrentConfigurationVersion</name>
      <dataType>ui4</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>SupportedDataModelsUpdate</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>SupportedParametersUpdate</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <Optional/>
      <name>AttributeValuesUpdate</name>
      <dataType>string</dataType>
    </stateVariable>
```

```
<stateVariable sendEvents="yes">
  <Optional/>
  <name>InconsistentStatus</name>
  <dataType>boolean</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_StructurePath</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_StructurePathList</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_PartialPath</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_ParameterValueList</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_NodeAttributeValueList</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_ParameterInitialValueList</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_Filter</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_SupportedDataModels</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_SearchDepth</name>
  <dataType>ui4</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_ChangeStatus</name>
  <dataType>string</dataType>
  <allowedValueList>
    <allowedValue>ChangesCommitted</allowedValue>
    <allowedValue>ChangesApplied</allowedValue>
  </allowedValueList>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_InstancePathList</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_ContentPathList</name>
  <dataType>string</dataType>
</stateVariable>

<stateVariable sendEvents="no">
  <name>A_ARG_TYPE_MultiInstancePath</name>
  <dataType>string</dataType>
</stateVariable>
```

```
    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_InstancePath</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_NodeAttributePathList</name>
      <dataType>string</dataType>
    </stateVariable>

  </serviceStateTable>
</s:scpd>
```

Appendix A: XML schema (Normative)

This appendix contains the XML normative schema to be used to check for the actions' argument correctness. The XML schema below defines also the formal grammar described in 2.3.1.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CMS-XSD [
   <!ENTITY Numeric "([0-9]|([1-9][0-9]+))">
   <!ENTITY Wildchar "#">
   <!ENTITY Slash "/">
   <!ENTITY NodeName "([\i-[:-]][\c-[:\.-]]*)">
   <!ENTITY LeafName "&NodeName;">
   <!ENTITY SingleInstanceNodeName "(&NodeName;&Slash;)">
   <!ENTITY MultiInstanceNodeName "(&NodeName;&Slash;)">
   <!ENTITY Instance "(&Numeric;&Slash;)">
   <!ENTITY InstanceAlias "(&Wildchar;&Slash;)">
   <!ENTITY InternalNode "(&SingleInstanceNodeName;|(&MultiInstanceNodeName;&Instance;))">
   <!ENTITY InternalAlias
"(&SingleInstanceNodeName;|(&MultiInstanceNodeName;&InstanceAlias;))">
   <!ENTITY RootPath "&Slash;">
   <!ENTITY ParameterPath "(&RootPath;&InternalNode;*&LeafName;)">
   <!ENTITY SingleInstancePath
"(&RootPath;|(&RootPath;&InternalNode;*&SingleInstanceNodeName;))">
   <!ENTITY MultiInstancePath "(&RootPath;&InternalNode;*&MultiInstanceNodeName;)">
   <!ENTITY InstancePath "(&RootPath;&InternalNode;*&MultiInstanceNodeName;&Instance;)">
]>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:cms="urn:schemas-upnp-
org:dm:cms" targetNamespace="urn:schemas-upnp-org:dm:cms" elementFormDefault="unqualified"
attributeFormDefault="unqualified" version="1-yyyymmdd">
   <xs:simpleType name="Path">
      <xs:restriction base="xs:token"/>
   </xs:simpleType>
   <xs:simpleType name="RootPath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&RootPath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="ParameterPath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&ParameterPath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="SingleInstancePath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&SingleInstancePath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="MultiInstancePath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&MultiInstancePath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="InstancePath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&InstancePath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="ParameterOrMultiInstancePath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&ParameterPath;"/>
         <xs:pattern value="&MultiInstancePath;"/>
         <xs:pattern value="&InstancePath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="PartialPath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&RootPath;"/>
         <xs:pattern value="&SingleInstancePath;"/>
         <xs:pattern value="&MultiInstancePath;"/>
         <xs:pattern value="&InstancePath;"/>
      </xs:restriction>
   </xs:simpleType>
```

```
   <xs:simpleType name="ContentPath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&RootPath;"/>
         <xs:pattern value="&SingleInstancePath;"/>
         <xs:pattern value="&MultiInstancePath;"/>
         <xs:pattern value="&InstancePath;"/>
         <xs:pattern value="&ParameterPath;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="StructurePath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&RootPath;(&InternalAlias;)*&LeafName;?"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="ParameterInitializationPath">
      <xs:restriction base="cms:Path">
         <xs:pattern value="&SingleInstanceNodeName;*&LeafName;"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="Value">
      <xs:simpleContent>
         <xs:extension base="xs:anySimpleType"/>
      </xs:simpleContent>
   </xs:complexType>
   <xs:element name="StructurePathList">
      <xs:annotation>
         <xs:documentation>Defines a list of StructurePaths.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="StructurePath" type="cms:StructurePath"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="ParameterValueList">
      <xs:annotation>
         <xs:documentation>Defines a list of Parameter elements. Each Parameter element is
a ParameterPath-Value pair.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Parameter">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="ParameterPath" type="cms:ParameterPath"/>
                     <xs:element name="Value" type="cms:Value"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="NodeAttributeValueList">
      <xs:annotation>
         <xs:documentation>Defines a list of Node elements. Each Node contains the
NodeAttributePath (type: ParameterOrMultiInstancePath) element and values for its
associated attributes.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Node">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="NodeAttributePath"
type="cms:ParameterOrMultiInstancePath"/>
                     <xs:element name="Type" minOccurs="0">
                        <xs:simpleType>
                           <xs:restriction base="xs:token">
                              <xs:enumeration value="string"/>
                              <xs:enumeration value="int"/>
                              <xs:enumeration value="long"/>
                              <xs:enumeration value="unsignedInt"/>
                              <xs:enumeration value="unsignedLong"/>
                              <xs:enumeration value="boolean"/>
                              <xs:enumeration value="dateTime"/>
```

```
                           <xs:enumeration value="base64"/>
                           <xs:enumeration value="hexBinary"/>
                        </xs:restriction>
                     </xs:simpleType>
                  </xs:element>
                  <xs:element name="Access" minOccurs="0">
                     <xs:simpleType>
                        <xs:restriction base="xs:token">
                           <xs:enumeration value="readWrite"/>
                           <xs:enumeration value="readOnly"/>
                        </xs:restriction>
                     </xs:simpleType>
                  </xs:element>
                  <xs:element name="Version" type="xs:unsignedInt" minOccurs="0"/>
                  <xs:element name="EventOnChange" type="xs:boolean" minOccurs="0"/>
                  <xs:element name="MIMEType" type="xs:token" minOccurs="0"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="ParameterInitialValueList">
   <xs:annotation>
      <xs:documentation>Defines a list of Node elements. Each Node element is a
ParameterInitializationPath-Value pair.</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
         <xs:element name="Node">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="ParameterInitializationPath"
type="cms:ParameterInitializationPath"/>
                  <xs:element name="Value" type="cms:Value"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="SupportedDataModels">
   <xs:annotation>
      <xs:documentation>Defines a list of SubTree elements. Each SubTree element
contains information about a supported data model.</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
         <xs:element name="SubTree">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="URI" type="xs:anyURI"/>
                  <xs:element name="Location" type="cms:SingleInstancePath"/>
                  <xs:element name="URL" type="xs:anyURI" minOccurs="0"/>
                  <xs:element name="Description" type="xs:string" minOccurs="0"/>
                  <xs:element name="SourceLocation" type="xs:string" minOccurs="0"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="InstancePathList">
   <xs:annotation>
      <xs:documentation>Defines a list of InstancePaths.</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:element name="InstancePath" type="cms:InstancePath"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
<xs:element name="ContentPathList">
   <xs:annotation>
      <xs:documentation>Defines a list of ContentPaths.</xs:documentation>
```

```
      </xs:annotation>
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="ContentPath" type="cms:ContentPath"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="NodeAttributePathList">
      <xs:annotation>
         <xs:documentation>Defines a list of NodeAttributePath (type:
ParameterOrMultiInstancePath) nodes used to retrieve attribute values.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="NodeAttributePath" type="cms:ParameterOrMultiInstancePath"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Appendix B: Common Objects (Normative)

This appendix specifies the basic data model for any CMS implementations, which is a list of *Parameters* maintained by the CMS that can be retrieved and, where applicable, changed by a control point. All CMS implementations SHALL provide all the required (R) *Parameters*. It's left to the implementations to provide also the optional *Parameters* (not required by this specification) and, if needed, custom extensions to the following data model.

Custom extension *Parameters* as well as data model offered by other UPnP services (whether they are part of UPnP DM or not) have to be defined in specific documents and are outside the scope of this CMS specification.

In general, *Parameter* may be used for: software management, configuration management, diagnostic and performance monitoring, as summarized in the following descriptions.

- Software management requires the description of the capabilities of the managed device. These capabilities are associated to the managed device and the firmware/software it maintains. Since they are may be associated to the hardware, they are not meant to change and they are not subject to third party configuration. They are often read-only *Parameters*.
- Configuration management concerns the configuration *Parameters* of the environment that are provided to the devices. The configuration adapts the application – delivered by the software (possibly firmware) installed on the device – to the surrounding context: network, time zone, device location, user identity and preferences. This topic requires the management of *Parameters* writable by (authorized) device management actors. Indeed, configuration management requires the ability to retrieve the current values of the available device *Parameters*, either configuration *Parameters* or status *Parameters*: values retrieved are usually needed in order to appropriately update the device configuration.
- Diagnostics is a function called punctually by the user or the device management system (i.e. the control point) at periodic time or at the time of dysfunctions detection. The diagnostics function is performed through the call of actions testing the capabilities or the applications of the device. 'Ping', 'traceroute' are diagnostics operations testing the networking capabilities of the device.
- Performance monitoring function continuously gathers statistics on the device usage (e.g., cpu usage, amount of free memory, application usage). Statistics concerns device *Parameters* that are frequently changing at runtime. The performance monitoring function is complementary to the diagnostics function. The diagnosis of problems on the device relies on both functions. Device diagnosis enables the Device Management system to take measures to face dysfunctions of the device. The semantics of diagnostics actions and the high frequency of the change of performance *Parameters* make these functions separate from Software management and configuration.

3.1. Reserved namespaces

In order to possibly avoid conflicts in data model definitions, some namespaces (i.e. common prefix *PartialPath* for *Parameters*) have been defined herein. This means that given a prefix for a data model as `/reserved/` and the data model containing the definition of *Parameter* names `p` and `m/#/f`, the resulting names for them are the concatenations: `/reserved/p` and `/reserved/m/#/f`.

Reserved prefixes are defined in the following table. When the reserved name cannot be defined a rule is recommended.

Table 0-39: Reserved PartialPaths and rules for prefixes

PartialPath	Description
/UPnP/DM/	Common prefix for all *Parameters* in the *Parent Device* data model as defined in this CMS document.
/UPnP/DM/Software/	Common prefix for all *Parameters* in the *Parent Device* data model as defined in this SMS document.
/UPnP/<device>/	Whenever an UPnP device defines its own data model, the WC moniker MUST be used for the <device> placeholder. Consequently all its *Parameters* MUST have the name beginning with such prefix *PartialPath*. For example: /UPnP/PHONE/ might be the common prefix for all *Parameters* defined by the UPnP Telephony Working Committee.
/…/X_<vendor>/	As stated in [1.7] vendor specific data models may be linked to any *Node* in the mandatory data model and MUST begin with X_ concatenated by the vendor domain name. In case of data model definition imported from another organization, it is also REQUIRED the use of X_ prefix. For example parameters in the data model which definitions are imported from the Broadband Forum should be prefixed by /X_BroadBand_Forum/

3.2. Configuration Management Service Data Model

All name in this table of *Parameter* definitions must be prefixed by `/UPnP/DM/`.

Columns' description:

- **Name**: white rows contain *Leaf* names, whereas yellow rows contain *StructurePath* fragments from the common prefix to the *SingleInstance* or *MultiInstance Node*.
- *Type*: the *Type* attribute value for *Leaf Nodes*, otherwise (yellow rows) it is specified whether the *Node* is *SingleInstance* or *MultiInstance*.
- *Acc.*: stands for *Access* attribute value of the *Node*. Possible values are "W" (the *Parameter* is writable, or the *Instance* is creatable) and "-" (the *Parameter* is read only). If a *Parameter* is writable means that it makes sense to write (i.e. configure) it, and therefore does not mean that it must be writable for all implementations. The control point should use the *GetAttributes()* action to verify what is implemented on the device. On the opposite side, if a *Parameter* is read only means that it does not make sense to write (i.e. configure) it, and therefore it must be read only for all implementations. Check with section 2.3.2.2 for further details concerning this attribute.
- **Req**.: stands for Required. Possible values are "R" (the *Node* implementation is required), "O" (the *Node* implementation is optional) and "CR" (the *Node* implementation is conditionally required).
- **Description**: describes the *Parameter* meaning.
- *EOC*: stands for *EventOnChange*. Indicates whether the *EventOnChange* attribute is supported by the *Node* and its default value. **Note:** Vendors can extend the list of the *Parameters* supporting the *EventOnChange* attribute.
- *Ver*: stands for *Version*. Indicates when the *Version* attribute is supported, whether the *Parameter* MUST also support (R) it. The dash "-" means that the support for that attribute is optional.

Name	Type	Acc.	Req	Description	EOC	Ver
/UPnP/DM/DeviceInfo/	*SingleInstance*	-	R	This is the DeviceInfo section of the data model, as mentioned thorough the CMS document and contains general device information.	-	-
FriendlyName	string(64)	W	O	FriendlyName in the Device Description, which is a writeable asset tracking identifier for the Device, i.e. a user friendly name for the device.	1	R
ProvisioningCode	string(64)	W	R	Identifier of the primary service provider and other provisioning information.	1	-
SoftwareVersion	string(64)	-	R	The current software version of the *Parent Device*.	1	R
SoftwareDescription	string(256)	-	R	Describes the software for which the SoftwareVersion applies.	1	-
UpTime	unsignedInt	-	R	Time in seconds since the *Parent Device* was started.	-	-
/UPnP/DM/DeviceInfo/PhysicalDevice/	*SingleInstance*		CR	Information related to the physical device. It MUST be provided when the *Parent Device* has access to the physical device.	-	-
ContactInfo	string(256)	-	O	This *Parameter* shows mail address / telephone number for inquires. The user can inquire for the error (hardware / application error, not network one).	-	-
Name	string(64)	W	O	User-assigned and writeable asset tracking identifier for the Device	1	R
OwnerName	string(64)	W	O	Name of the principal owner of the device.	-	-
Location	string(256)	W	O	A free-form string indicating the physical location of the device.	-	-
HardwareVersion	string(64)	-	R	A string identifying the particular hardware model and version supporting the ManageableDevice . This value may be empty if such information is not available to the UPnP CMS.	-	-
NetworkInterfaceNumberOfEntries	unsignedInt	-	R	Number of instances of network interfaces.	1	R
/UPnP/DM/DeviceInfo/PhysicalDevice/DeviceID/	*SingleInstance*		R	Unique physical device identifier. The triplet {ManufacturerOUI, ProductClass, SerialNumber} MUST be guaranteed unique by the device vendor at manufacturing time. This value MUST remain fixed over the lifetime of the device, including across firmware updates.	-	-
ManufacturerOUI	hexBinary(3:3)	-	R	Organizationally unique identifier of the device manufacturer. The format is available at the following link: http://standards.ieee.org/regauth/oui/index.shtml.	-	-
ProductClass	string(64)	-	R	Identifier of the class of product for which the serial number applies. This may be the same as in ModelName or ModelNumber defined in DDD.	-	-
SerialNumber	string(64)	-	R	Serial number of the physical device. If SerialNumber is also present in the DDD, it MUST have the same value.	-	-
/UPnP/DM/DeviceInfo/PhysicalDevice/NetworkInterface/#/	*MultiInstance*		R	Information related to the Physical Network Interfaces available on the device.	-	-
SystemName	string(64)	-	R	Unique key. This is the name provided by the underlying system to the network interface.	-	-

Name	Type	Acc.	Req	Description	EOC	Ver
Description	string(256)	-	O	Textual description of the interface. It should contain the hardware description of the network interface card.	-	-
MACAddress	string(17)	-	R	The MAC address of the physical interface.	-	-
InterfaceType	string	-	R	Type of this physical interface. Enumeration of: “Ethernet” “USB” “802.11” “HSDPA” “HomePNA” “HomePlug” “MoCA” “G.hn” “UPA” “Other”	-	-
/UPnP/DM/DeviceInfo/OperatingSystem/	*SingleInstance*		CR	Information related to the operating system. It MUST be provided when the *Parent Device* has access to the operating system.	-	-
SoftwareVersion	string(64)	-	R	A string identifying the version of the operating system.	1	R
SoftwareDescription	string(256)	-	R	Describes the software for which the SoftwareVersion applies. The format is vendor specific and might contain, for example, information concerning the operating system name, the version, the name of the implementation, the version of this implementation, the type of the underlying processor and so on.	1	-
UpTime	unsignedInt	-	R	Time in seconds since the operating system has been started,	-	-
LastUpgradeDate	dateTime	-	O	Date of installation or of the last upgrade of the operating system.	-	-
WillReboot	boolean	-	R	Indicates whether the BMS::Reboot() will reboot the operating system.	-	-
WillBaselineReset	boolean	-	R	Indicates whether the *BMS::BaselineReset()* will reset the operating system and other system level resources and settings.	-	-
/UPnP/DM/DeviceInfo/ExecutionEnvironment/	*SingleInstance*		CR	Information related to the targeted Execution Environment [SMS]. It MUST be provided when the Parent Device has access to targeted Execution Environment and the Execution Environment is not the Operating System.	-	-

Name	Type	Acc.	Req	Description	EOC	Ver
Status	string	-	R	Current operational status of the targeted Execution Environment [SMS]. Allowed values are: "Initializing" "Up" "Up_but_about_to_reboot" : sub-state of UP	1	R
UpTime	unsignedInt	-	R	Time in seconds since the Execution Environment [SMS]. has been started,.	-	-
SoftwareVersion	string(64)	-	R	A string identifying the software/firmware version of the running Execution Environment.	1	R
SoftwareDescription	string(64)	-	R	Describes the targeted Execution Environment [SMS]. for the *ManageableDevice*. The format is vendor specific and might contain, for example, information concerning the execution environment standard name, the version of the standard, the name of the implementation, the version of this implementation, the type of the underlying processor and so on.	1	-
LastUpgradeDate	dateTime	-	O	Date of installation or of the last upgrade of the Execution Environment [SMS].	-	-
WillReboot	boolean	-	R	Indicates whether the BMS::Reboot() will reboot the Execution Environment [SMS].	-	-
WillBaselineReset	boolean	-	R	Indicates whether the *BMS::BaselineReset() will* reset the Execution Environment [SMS] and other system level resources and settings..	-	-
/UPnP/DM/Configuration/	*SingleInstance*		R	Information related to the state of the device. The *Parameters* available in this sub-tree are the one a control point may want to modify in order to update the device's state or behavior.	-	-
/UPnP/DM/Configuration/Network/	*SingleInstance*		CR	Information related to the networking configuration. A control point will find here a means to configure the IP stack of the device. It MUST be provided when the Parent Device has access to the network configuration.	-	-
HostName	string(64)	W	R	The host name of the device, which can be provided to a DHCP server and registered with a DNS server.	1	R
IPInterfaceNumberOfEntries	unsignedInt	-	R	Number of IP interface instances.	-	-
/UPnP/DM/Configuration/Network/IPInterface/#/	*MultiInstance*		R	Each instance of this sub-tree stands for an IP network interface identified by its system name. Each interface can be configured independently.	-	-
SystemName	string(64)	-	R	Unique key. This is the name provided by the underlying system to the IP interface.	0	R
/UPnP/DM/Configuration/Network/IPInterface/#/IPv4/	*SingleInstance*		R	Data related to the IPv4 stack configuration.	-	-
IPAddress	string	W	R	The current IP address assigned to this interface.	1	R
AddressingType	string	W	R	The method used to assign an address to this interface. Enumeration of: "DHCP" "Static" "AutoIP"	0	R
DNSServers	string(256)	W	R	Comma-separated list of IP address of the DNS servers for this interface.	0	R

Name	Type	Acc.	Req	Description	EOC	Ver
SubnetMask	string	W	R	The current subnet mask.	0	R
DefaultGateway	string	W	R	The IP address of the current default gateway for this interface.	0	R
/UPnP/DM/Configuration/Network/IPInterface/#/IPv6/	*SingleInstance*		O	Data related to the IPv6 stack configuration.	-	-
DNSServers	string(256)	W		Comma-separated list of IPv6 address of the DNS servers for this IP interface.	0	R
DefaultGateway	string	W		The IPv6 address of the current default gateway for this IP interface.	0	R
IPv6AddressNumberOfEntries	unsignedInt	-	R	Number of entries in the IPv6 addresses table.	1	-
/UPnP/DM/Configuration/Network/IPInterface/#/IPv6/Address/#/	*MultiInstance*		R	IPv6 addresses configuration.	-	-
IPAddress	string	W	R	This shows current IPv6 address for each IPv6 interface.	1	R
IPAddressType	string	W	R	The type of the address. Enumeration of: “GlobalAddress” “LinkLocalAddress” “SiteLocalAddress”	0	R
AddressingType	string	W	R	The method used to assign an address to this interface. Enumeration of: “DHCP” “Static” "RA"	0	R
Prefix	string	W	R	The current prefix used for the IPv6 address.	0	R
Temporary	boolean	W		A flag to determine if the IP address is temporary.	0	R
AddressStatus	string	-	O	The current status of this address. Enumeration of: “Tentative” “Preferred” “Valid” “Invalid”	1	-
/UPnP/DM/Monitoring/	*SingleInstance*		R	This sub-tree contains the usage information related to resources available on the device.	-	-
NetworkUsageNumberOfEntries	unsignedInt	-	R	Number of entries in the NetworkUsage table.	1	-
StorageNumberOfEntries	unsignedInt	-	R	Number of entries in the Storage table.	1	-
/UPnP/DM/Monitoring/OperatingSystem/	*SingleInstance*		CR	Usage status of the available operating system resources. It MUST be provided when the Parent Device has access to the operating system.	-	-
CurrentTime	dateTime	-	R	The current system date and time.	-	-

Name	Type	Acc.	Req	Description	EOC	Ver
CPUUsage	unsignedInt [0:100]	-	R	The total amount of the CPU currently being used rounded up to the nearest whole percent.	-	-
MemoryUsage	unsignedInt [0:100]	-	R	The total amount of the memory currently being used rounded up to the nearest whole percent.	-	-
/UPnP/DM/Monitoring/ExecutionEnvironment/	*SingleInstance*		CR	Usage status of the available Execution Environment [SMS] resources. It MUST be provided when the Parent Device has access to the operating system and the Execution Environment is not the Operating System.	-	-
CPUUsage	unsignedInt [0:100]	-	R	The total amount of the CPU currently being used by the Execution Environment [SMS] rounded up to the nearest whole percent.	-	-
MemoryUsage	unsignedInt [0:100]	-	R	The total amount of the memory currently being used by the Execution Environment [SMS] rounded up to the nearest whole percent.	-	-
/UPnP/DM/Monitoring /IPUsage/#/	*MultiInstance*		CR	IP interface status and throughput statistics. It MUST be provided when the Parent Device has access to the network statistics information.	-	-
SystemName	string(64)	-	R	Unique key. Value of the corresponding IP interface's /UPnP/DM/Configuration/Network/IPInterface/#/SystemName parameter.	-	-
Status	string	-	R	Status of the IP interface. Allowed values are: "UP", "DOWN".	1	R
TotalPacketsSent	unsignedInt	-	R	Total number of IP packets sent over this IP interface since the interface last came up.	-	-
TotalPacketsReceived	unsignedInt	-	R	Total number of IP packets received over this IP interface since the interface last came up.	-	-
/UPnP/DM/Monitoring/ Storage/#/	*MultiInstance*	-	CR	Status of the device storage (e.g. Flash memory, Disks). This *Parameter* doesn't want to interfere with the Storage WC, and can be used, for example, in trouble-shooting where there is not enough space to play a media content.	-	-
PointNode	string	-	R	System path of the mount point where the storage is mounted on.	-	-
Usage	unsignedInt [0:100]	-	R	The total amount of the disk space currently being utilized rounded up to the nearest whole percent.	-	-

Appendix C: Mapping rules for Other Organizations (Informative)

Rules for mapping an organization's data model to UPnP DM have to be specified independently for each organization.

Note that, in order for it to be possible to use data models defined by other organizations, it is necessary that CMS actions and concepts map well to the actions and concepts envisaged by those other organizations. For example, BBF data models are defined to work with TR-069, so it is important that CMS actions and concepts map well to TR-069 data model operations and concepts. Similarly, OMA data models are defined to work with OMA-DM, and MIBs are defined to work with SNMP. Therefore, the data model mapping rules MUST also consider the mapping of protocol operations and concepts.

This section presents a fairly complete set of BBF (TR-069) mapping rules, and an outline of possible OMA (OMA-DM) and MIB (SNMP) mapping rules.

3.3. BBF (TR-069) Mapping Rules

These rules are divided into the following categories:

- **Name:** rules for mapping BBF object and *Parameter* names to UPnP DM names (rules are to be applied in order). Note that UPnP DM name rules are similar to BBF ones allowing any name that is a valid XML NCName (no-colon name) except that (for obvious reasons) it doesn't permit dots and hyphens "-".
- **Type**: rules for mapping BBF data types to UPnP DM data types.
- **List:** rules for mapping BBF lists to UPnP DM lists.
- **Reference:** rules for mapping BBF data model references to UPnP DM data model references.

ID	Category	Description
1	Name	If name begins with dot, remove it. The current CMS document does not describes relative paths, but the obvious syntax is that a path that starts "/" (i.e. the *Root*) is absolute and that all other paths are relative. Therefore (recall that all non-*Leaf Node* names end with "/"), a full path for CMS is just the concatenation of a partial path and a relative path, as in "/BBF/" + "STBService/XXX/" must be transformed in "/BBF/STBService/XXX/".
2	Name	If name begins with `Device.` or `InternetGatewayDevice.`, remove it (including the dot).
3	Name	If name begins with `Services.`, remove it (including the dot).
4	Name	Replace dot separators with slashes.
5	Name	Replace "{i}" placeholders with "#". The "#" symbol is used in two contexts: (a) to indicate in the data model description that an object is multi-instance and (b) when actions are used to manage the data model, to represent the concept of "all" instance *Nodes*.

ID	Category	Description
6	Type	No mapping necessary, except that if BBF definition uses a named type, such as `IPAddress`, this is treated as a textual convention, e.g. `IPAddress` would be treated as "string, format xxx, representing IP address".
7	List	No mapping necessary, because comma-separated list are considered as string in CMS.
8	Reference	For relative references (references that are within the BBF data model definition), all the above name mapping rules apply. In addition, append a slash (if necessary) to non *Leaf Node* references (BBF object references are not dot-terminated).
9	Reference	For absolute references (references outside the BBF data model definition), it is not possible to give a general rule. Such references are rare, but occasionally a *Parameter* might reference something in a common object (e.g. in DeviceInfo), or there might be a reference to another Service object (e.g. TR-135 STBService instances can reference TR-140 StorageService instances). If such a requirement arises, the requirement must be stated in plain English, e.g. in the following (taken from TR-135 and translated following UPnP DM grammar rules): "References the corresponding StorageService instance, or an object contained within such an instance, e.g. a PhysicalMedium, LogicalVolume or Folder instance. The value is the full hierarchical name of the corresponding object. Example: Device/Services/StorageService/1".

TR-069 data model operations and concepts already map well to CMS actions and concepts. CMS instance numbers may start at 0 therefore TR-069 proxies should map them by adding 1 to go from CMS to TR-069 and by subtracting 1 to go the other way.

3.4. OMA (OMA-DM) Mapping Rules

These rules are in draft version. Further improvement could be provided in subsequent versions of this *ConfigurationManagement:1* Service Template:

ID	Category	Description
1	Name	If name begins with dot, remove it. "/" is considered to be the *Root Node* in CMS.
2	Name	CMS never uses absolute path names; it always uses names relative to the *Root Node*. The leading "`./`" in OMA names is always omitted.
3	Name	*Path* names don't end with "/" in OMA. So add a trailing "/" to these names.
4	Property	Type. All *Nodes* have a Type property in OMA which corresponds to the optional *MIMEType* attribute in CMS. The *Type* attribute of a *Leaf Node* is always the MIME type of the current object value. The *Type* property of interior *Nodes* is either a Management Object Identifier URI or it has no value.

ID	Category	Description
5	Property	Optionally OMA DM *Nodes* have a "Title" property which can be used by the Server to assign a human readable alias to a *Node*. This will be ignored by the CMS.
6	Data Type	XML data of *Leaf Nodes*, to be treated as string input for CMS.
7		There is no explicit concept of table, or of unique key and the grammar extension has to be specified.
8		There is no concept of instance number in OMA.

3.5. MIB (SNMP) Mapping Rules

These rules are in draft version. Further improvement could be provided in subsequent versions of this *ConfigurationManagement:1* Service Template:

- SNMP doesn't use path names as such, but a hierarchy can be inferred by (a) regarding objects not in tables as being at the top level, (b) regarding tables with index columns that are all within the table as being top-level tables, (c) regarding tables with index columns that are in other tables as being either top-level tables with additional index columns (necessary if the external indices are not all in the same table), or else nested within the table that contains the external indices (possible only if all external indices are in the same table).
- FYI there is an unofficial (private) BBF tool that can convert a MIB definition into a BBF DM XML document. It does not implement all of the above logic, but it easily could do, and it acts as a proof of concept.
- SNMP doesn't support instance numbers. Instead, table rows are always accessed via index (key) value.
- Mandatory SNMP operations map well to UPnP DM ones (except that there is no `CreateAnd-Set()` operation).
- would be an implicit unique key in tables, so could always reference rows via `{name}`, as at present; similarly for SNMP).