SD2-6 Versioning And Extensibility

Abstract

This supporting document addresses versioning and extendibility mechanisms adopted by MTOSI.

Through the versioning mechanism it is possible to evolve the interfaces in a controlled manner maintaining backward and forward compatibility for a class of changes considered minor.

The extendibility mechanism allows a vendor (and\or an MTOSI architect) to tailor the specification to deal with future or specific concerns not addressed in the most current MTOSI release (latest version of the service interfaces).

Table of Contents

1 Introduction

Versioning entails rules and conventions for updating and supporting the MTOSI XML from one version to the next. Two classes of versions are identified each with different compatibility implications: minor and major versions. This contribution describes the mechanism for MTOSI versioning in order to guarantee the compatibility of minor versions and the possible migration strategies between major versions.

Extensibility mechanism is also addressed in order to allow a vendor (and/or an MTOSI architect) to extend the MTOSI specification with structures needed in any future or specific MTOSI deployment.

Detailed XML Interface Design aspects of versioning and extensions are covered in the Framework SD0-5_mTOPGuidelines_WebServices.

2 Forward and Backward Compatibility

MTOSI defines a set of interfaces exposed as services. As MTOSI specifications, these interfaces are called as “service interfaces”.

An ***interface*** consists of a set of ***operations***, which in turn consist of a set of ***messages***

E.g., **getTerminationPoint** operation in the **TerminationPointRetrieval** service interface of the ManagedResourceInventory DDP has:

- **getTerminationPointRequest**, the request message
- **getTerminationPointResponse,** the response message

All service interface messages are XML instance documents with associated XSDs (XML Schema Definition) for validation.

A **Service** exposes one or more service interfaces (wsdl:portType and related bindings). A **message processor** processes an XML instance. In Client-Server architectures both Client and Server or Sender and Receiver in a Request/Response business activity will be processing an XML instance. For this reason Backward and Forward compatibility is defined in relation to the processor role in which a sender or a receiver is acting [ORC01].

Backward compatibility – The message processor works correctly when receiving an old version of a message.

Forward compatibility – The message processor works correctly when receiving a new version of a message.

Service interfaces versions are **fully compatible** if the versions are **both backward and forward** compatible.

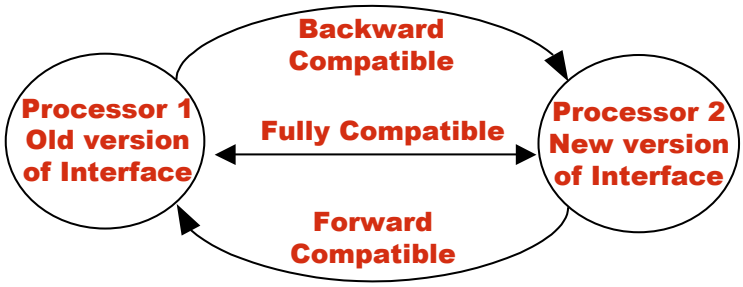

Figure 1 - Backward and Forward Compatibility

A **major update** is a change for which no meaningful communication between Sender and Receiver is possible with respect to the change, where the Sender and Receiver are using different versions. A message processor will not be able to natively process a message after a major update. After a major update the service will loose its backward compatibility (by definition).

A **minor update** is a change after which it is still possible for a processor to natively process a message. After a minor change the service will still be backward compatible.

3 Minor and Major Version Compatibility

3.1 Version Identifiers

Version identifiers are of the form “N.x”, where “N” represents the major version number and “x” represents the minor version string.

“N” must be a number

“x” must start with a number, but may also contain additional periods and numbers, e.g., “0”, “0.1”, “1.1”

Starting with version 1.0, each time a minor change is applied to the service interface the minor string is changed. The new minor version string should be lexically greater than the previous.

Each time a major change is introduced in a service interface, its major version number should be incremented.

3.2 Service Interface Versioning

In MTOSI, service interfaces and data models support both major and minor versioning as defined in the previous section.

Only the addition of new optional elements or optional structures in the XML Schema is both forward and backward compatible

Considering that compatible versions are classified as minor versions, the above axiom has the following implications:

- Adding a new optional interface to an MTOSI release is considered a minor change.
- Adding a new optional operation/notification to an interface is considered a minor change.
- Adding a new optional formal argument in an operation is considered a minor change.
- Adding a new optional element or structure to an MTOSI network object (TMF608) is considered a minor change.
- Adding or refining enumeration values as a minor change in a new release may be achieved by introducing an optional enumeration variable that specializes the base variable. Adding a new enumeration value directly in the enumeration set will result in a major version.
- Adding mandatory artifacts will result in a major version.
- Deleting artifacts by deprecating them will result in a minor version
- Deleting mandatory artifacts by removing them will result in a major version
- Deleting optional artifacts by removing them will result in a minor version

All **minor versions** of a service interface within a **major version** are defined to be compatible with each other

- E.g., 1.0, 1.0.1, 1.1, 1.2 are compatible with each other.

It should be noted that minor versions of a service interface are generally NOT identical but just compatible. Version 1.8 has different capabilities than 1.1. These differences make them different even though the addition of minor updates is still compatible. Any business activity supported by V1.1 should be supported by V1.8 with reasonable behaviors. The vice versa is also true. Note that the adjective "reasonable" refers to a contained semantic difference that can be tolerated in the context of a business activity.

This document includes the details on how to achieve interoperability between two MTOSI services implemented from two different minor versions of the same service interfaces major version (See).

Different **major versions** of a service interface are not compatible with each other

- E.g., 1.x and 2.x are not compatible versions

Although not normative, this document highlights several possible migration strategies to bridge MTOSI services implemented from two different major versions of the same service interface (See).

Refer to [SD0-5] for the details about the MTOSI XML specifications design requirements regarding versioning of the MTOSI service interfaces and data modules.

3.3 MTOSI Releases and Service Interface Versions

Each MTOSI service interface and data module has a distinct version number. All MTOSI service interfaces and data modules that are offered in MTOSI Release 2.0 have their version number set to 1.0 (See [SD0-5]). This independence of versioning allows each specification unit to evolve without impacting all the others.

It is important to note that:

- The versioning mechanism of the MTOSI service interfaces and data modules is not related to the MTOSI Release number.
- The MTOSI Document Delivery Packages (DDPs), which are specification components of an MTOSI Release, are not versioned.

The following example provides a simulation of the evolution of MTOSI Releases and their contents in the future:

- **MTOSI R2.0** has the following DDPs and service interfaces:
  - **RTM DDP**:
    - ***AlarmRetrieval*** Version 1.0
    - ***AlarmControl*** Version 1.0
  - **MRI DDP**:
    - ***ResourceInventoryRetrieval*** Version 1.0
    - ***TerminationPointRetrieval*** Version 1.0

- Future **MTOSI R2.1** may have (example only):
  - **RTM DDP**:
    - ***AlarmRetrieval*** Version 1.0 (unchanged)

        - ***AlarmControl*** Version 1.1 (minor change – new optional operations/attributes/parameters)
    - **MRI DDP**: (unchanged)
        - ***ResourceInventoryRetrieval*** Version 1.0
        - ***TerminationPointRetrieval*** Version 1.0

- Future **MTOSI R3.0** may have (example only):
    - **RTM DDP**:
        - ***AlarmRetrieval*** Version 2.0 (major change)
        - ***AlarmControl*** Version 1.1 (unchanged)
    - **MRI DDP**: (unchanged)
        - ***ResourceInventoryRetrieval*** Version 2.0 (major change)
        - ***TerminationPointRetrieval*** Version 1.0 (unchanged)

3.4 Achieving Minor Version Interoperability

A Message processor knows and supports a specific version of the service interface (major and minor number).

All messages with the same major version as the processor are compatible regardless of the minor version.

In this section we address the mechanism to achieve backward and forward compatibility in a minor version release.

3.4.1 Backward compatibility

By only adding optional element in an XML Schema (minor version change), the XML instances are backward compatible. A new version of a message processor will by default ignore any missing optional element in an old message instance.

3.4.2 Forward compatibility

Forward compatibility is exercised when an old processor receives a new version message. There are two main mechanisms to achieve forward compatibility: planning for extensions and Validation by projection.

3.4.3 Versioning Mechanism Options

3.4.3.1 Planned extensions

The XSD has a construct “ANY” that allows extending a schema maintaining forward compatibility. Nevertheless this construct has some side effects: it often introduces additional complexity (wrappers, namespaces) to deal with the intrinsic non-determinism, and it is often not realistic to know where and how an interface will be extended in the future. The “ANY” mechanism is best suited for well planned extensions such as the customer extensions under the UML vendorExtensions attribute.

3.4.3.2 Validation by projection

This technique advocates removing (projecting) all the unknown elements and structures not recognized in the message processor version, prior to validate and process the XML remains. [BAU04][ORC04].

In this way all the optional future tags added in minor release will be silently ignored by a current or earlier version message processor. While some of the current tools in the market allows this relax validation (e.g. Java XMLbeans), we can always achieve forward compatibility by pre-processing the XML instances before they reach the message processor. For example, an XSLT can be defined to pass through only those elements valid for a version of an interface. The same result can be achieved by materializing the XML message it a generic DOM structure and using for example XPATH to process the XML and only reference known elements. The implementation particulars of the above mentioned techniques are beyond the scope of MTOSI.

3.4.4 MTOSI Minor Version Mechanism

In MTOSI we adopt the "planned extensions" to address the vendor extensibility dimension and the "validation by projection" technique to address the forward compatibility of MTOSI service interfaces with the same major version.

In a nutshell, if a message minor version of an incoming message is greater than the processor's supported minor version, the processor **MUST silently ignore** extra elements it does not know (as note previously, these must be optional elements).

Refer to the "Advanced versioning features" section (not normative in MTOSI) in this document for some thoughts on how to modify this default behavior. Alternatively, the processor could add to the response message a list of all ignored tag elements in a warning containment.

If a service interface message (request, response, or fault) is validated against the associated MTOSI XML Schema, the XSLT mapping must be used prior to the validation step.

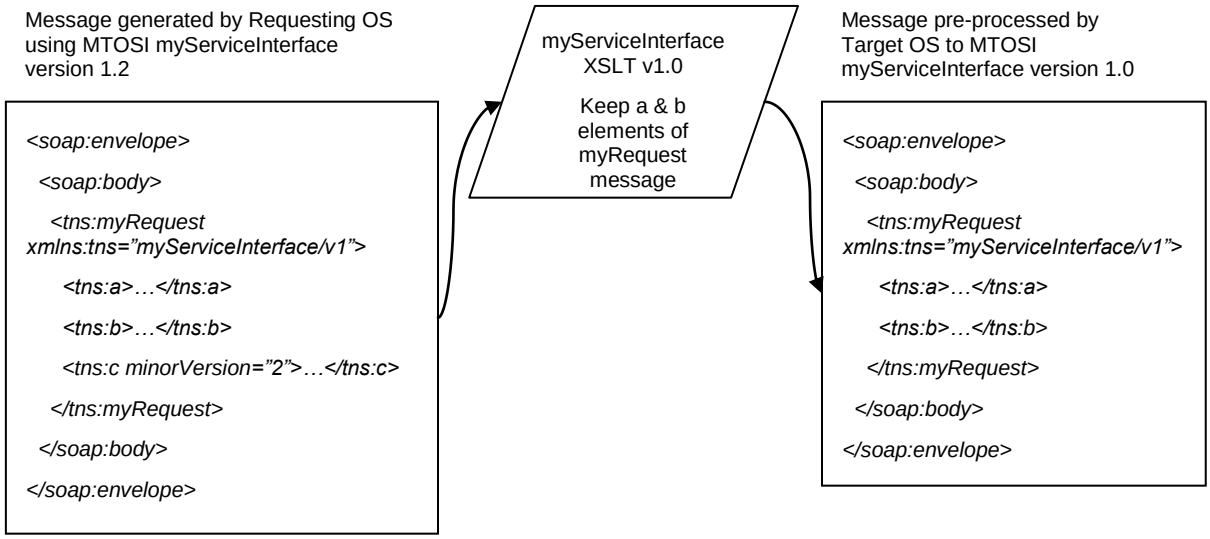

Figure 2 - Filtering the unknown elements

Refer to examples captured in Framework DDP in IIS/xml folder:

- Md.xml – an XML document instance based on MTOSI MD object version 1.0 (See Md.xsd in xsd folder)
- Md1-1.xsd – an example of the XML Schema for MTOSI MD object version 1.1
- Md1-1.xml – an XML document instance based on MTOSI MD object version 1.1 (see above XML Schema module example Md1-1.xsd)

3.4.5 Use Cases

Figure 3 shows a request/reply flow in the case both the client and processor support the exact same version.

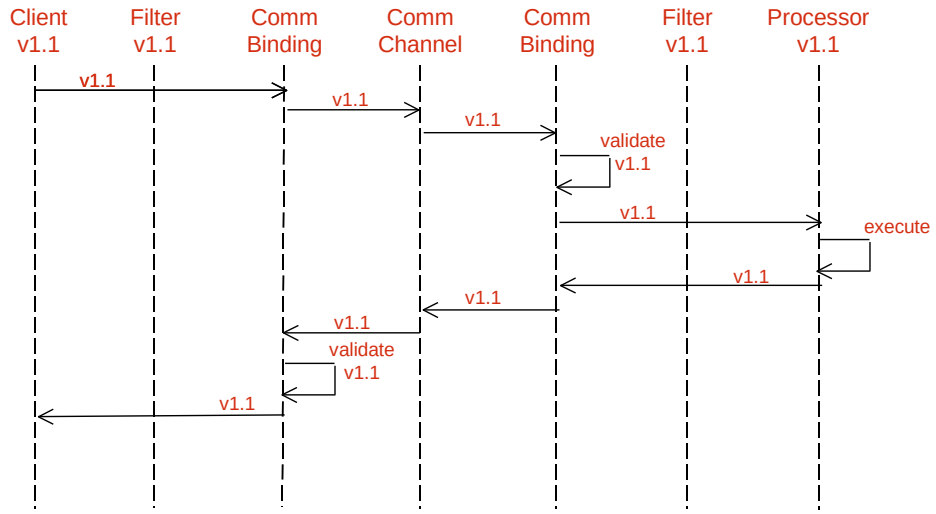

Figure 3 - Versions supported by client and processor match

Figure 4 shows a request/reply flow when the processor support a latter minor version than the client. Note that the filter on the client side transforms the v1.2 response into a v1.1 message, and the Communication Binding function on the processor side validates the v1.1 message against the v1.2 specification.

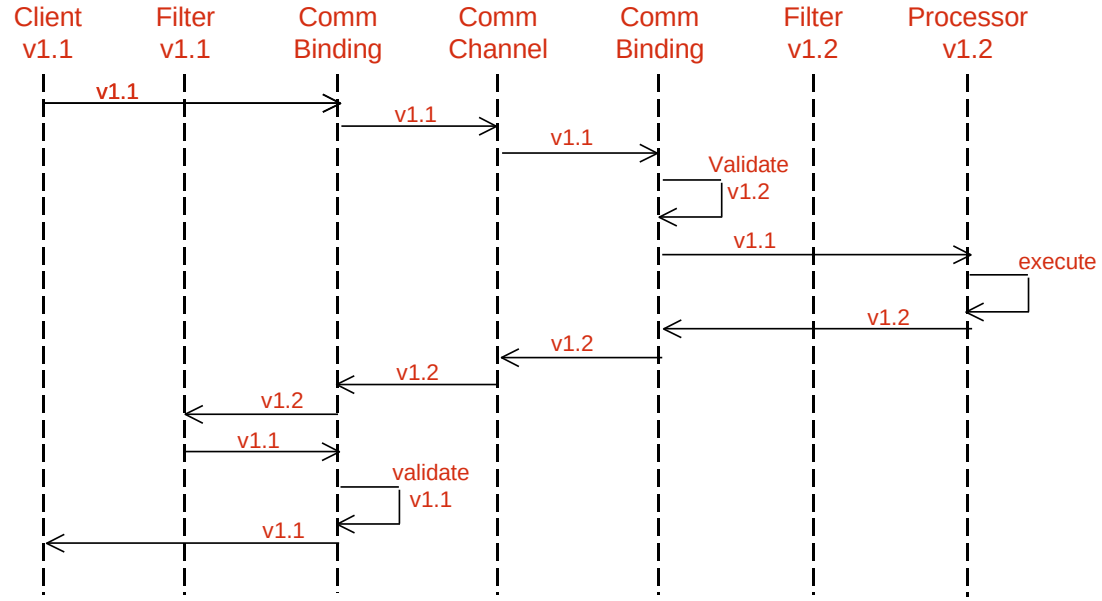

Figure 4 - Version supported by client older than that of processor

Figure 5 shows a request/reply flow when the client has a latter minor version than the processor. The filter on the processor side maps the message to v1.1 message and then the message is validated against the v1.1 XML and sent onto the processor. On the other side, the v1.1 message is validated against the v1.2 XML and then sent onto the client.

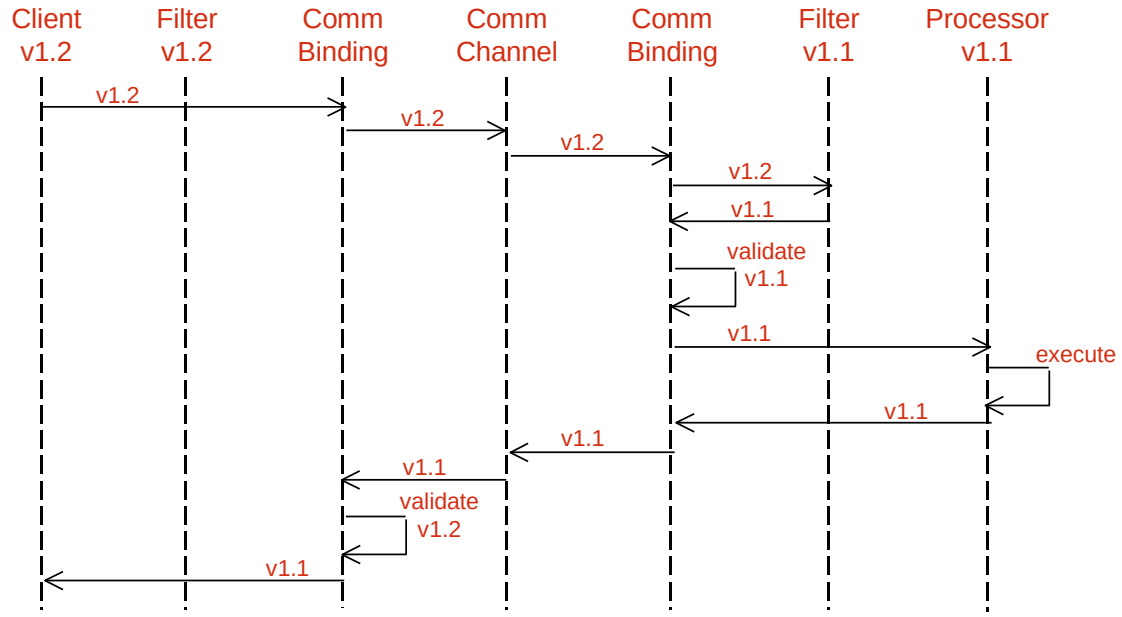

Figure 5 - Version supported by client newer than that of processor

The flows suggested in this section are not normative. They simply are meant to suggest a method for dealing with interoperability between MTOSI minor versions.

3.5 Major Version Interoperability Transition

All content in this section is not normative, but can be used as guidelines to dealing with interoperability of MTOSI clients and servers that implement an MTOSI service interface with two different major versions.

Both Client and Server processors should have some degree of independence when upgrading. When a client or server upgrades to a new major version of an interface, the previous major version should also be supported. A major version of an interface supported by a client must be available from a server.

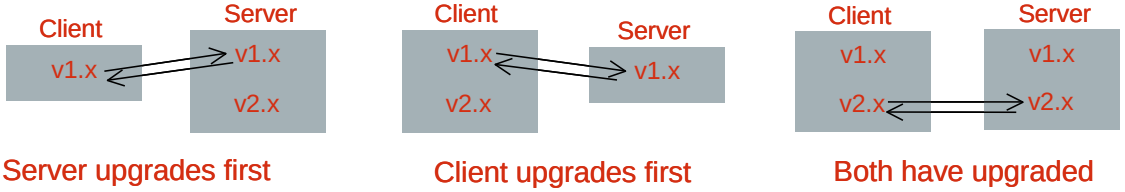

Figure 6 - Configurations showing multiple interface endpoints in clients and applications

Transition can be also accomplished through a mediation layer.

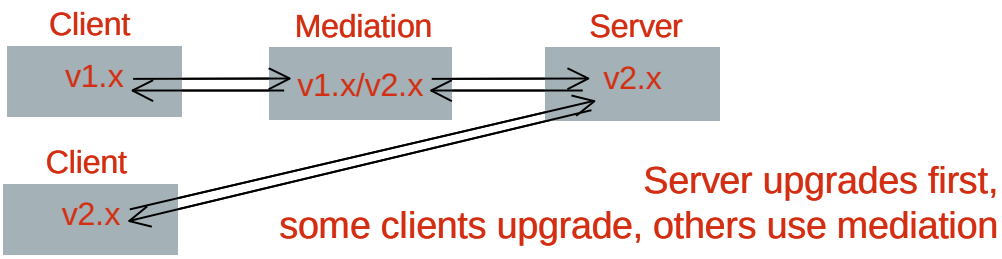

Figure 7 - Transition through a mediation layer

Notifications must be published in the major interface versions expected by subscribers.

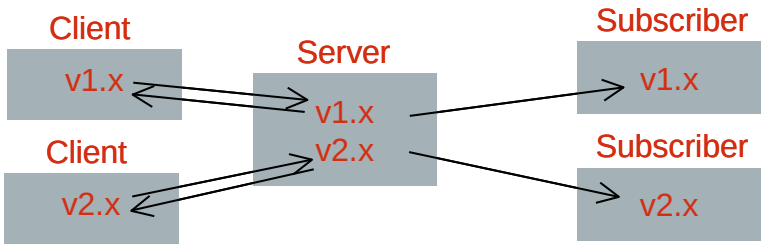

Figure 8 - Example of transition configuration with clients and subscribers on different major versions

If a server cannot support multiple major versions, mediation will also be required to generate notifications.

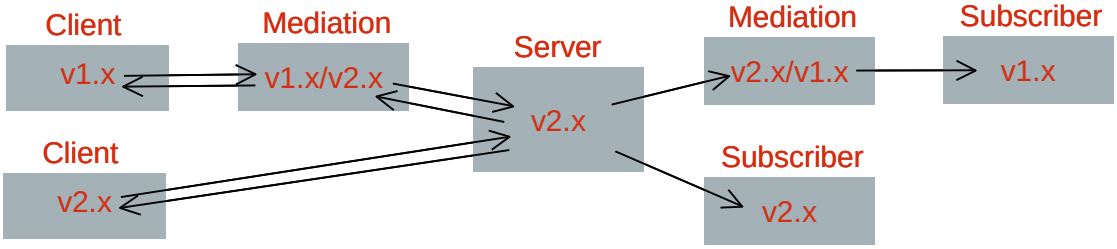

Figure 9 - Mediation layer for notifications

3.5.1 Use Cases

The following diagram in Figure 10 shows how a Server may support two major versions with two interface endpoints offered through a common communication binding stack. Both old and new clients can access the proper service by routing the request to the proper communication channel.

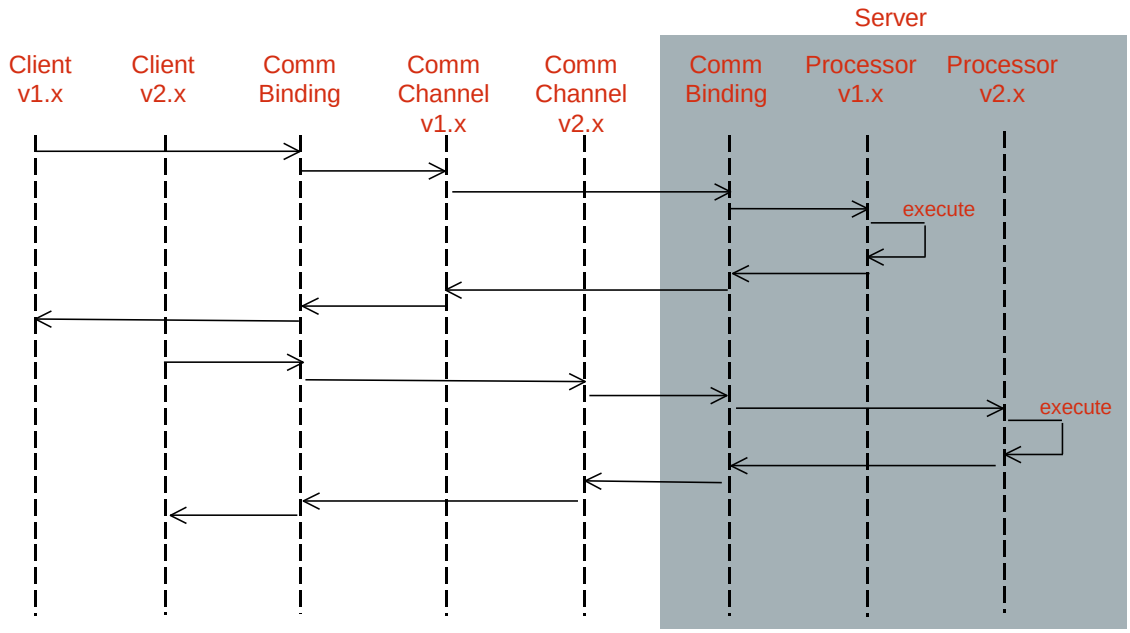

Figure 10 - Server supports two major versions with two interface endpoints, both old and new clients (Filters and validation not shown)

In the following diagram (Figure 11) a Server supports only one major version and uses a mediator layer during transition (Communication Binding not shown).

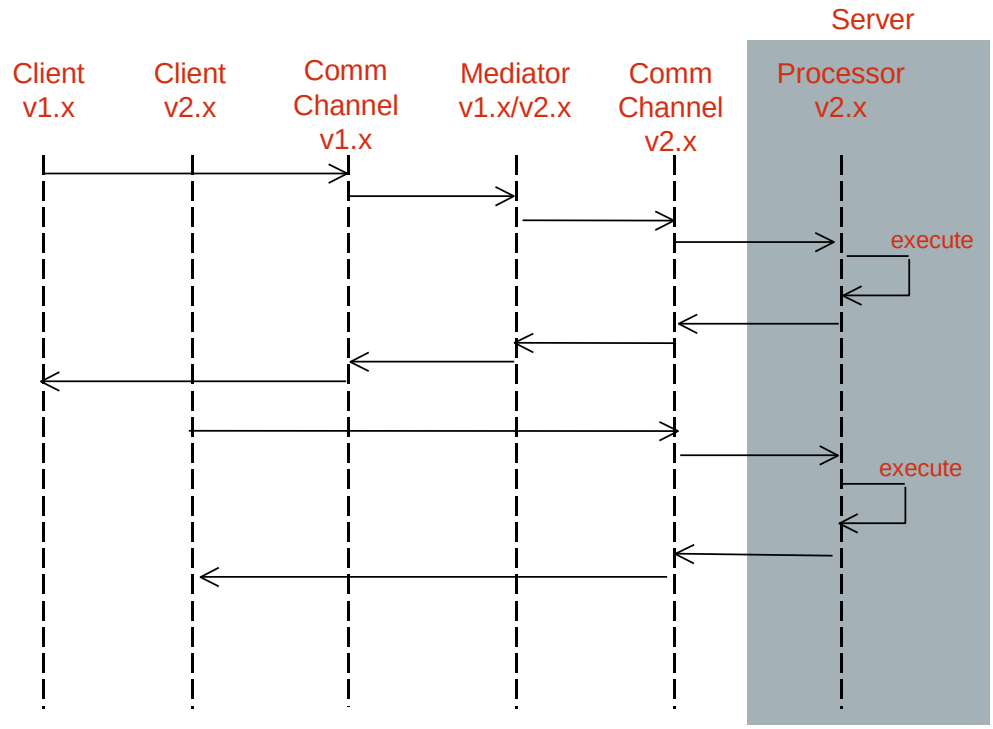

Figure 11 - Server supports only one major version, requires mediator during transition

The following diagram (Figure 12) shows how different communication channels and notification channels may be used during a transition from different major versions.

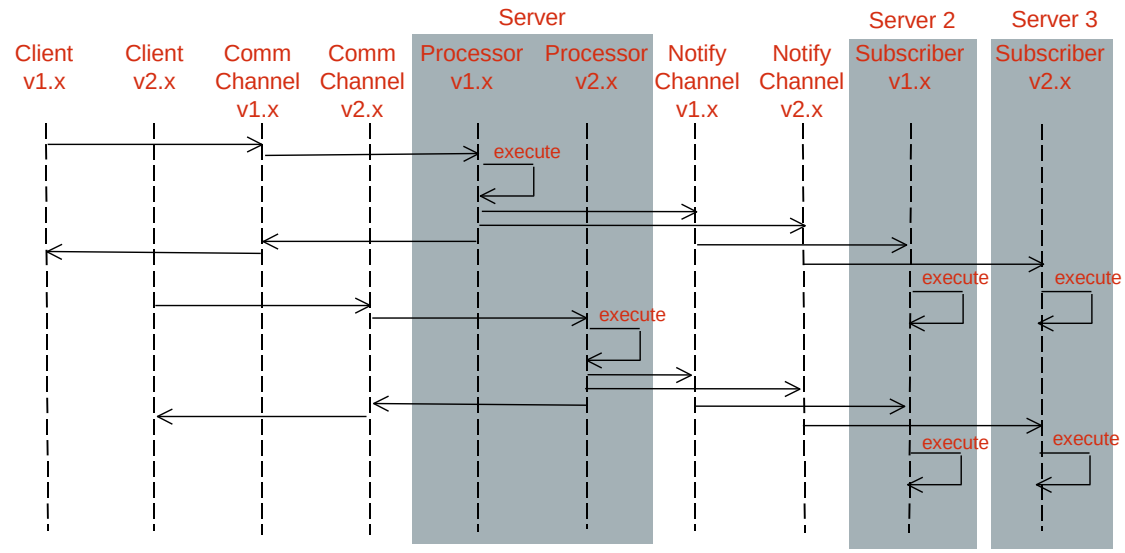

Figure 12 - Notification when there are subscribers on different major versions. Publication and subscription to Notify channels v1.x and v2.x is configurable

4 XML Schema Extensions

Extensibility is one of the essential design characteristics that are applied in the definition of the MTOSI XML Schema. Extensibility requirements are captured in the framework BA. The design of extension handlers in the MTOSI XML Schema is based on the following two considerations:

- Allow vendor to customize by extension the interface definitions, and
- Allow evolution of the interface definitions with minimal impact on compatibility.

The MTOSI XML Schema extension capabilities are covered in the following three sections:

- Complex Type Extension such as MTOSI objects and notifications (events)
- Simple Type Extensions such as MTOSI object attributes and parameters
- Vendor objects and notifications

Additionally, some XML examples are provided in the DDPs IIS XML directory.

4.1 XML Complex Type Extension

This section describes the adopted extension mechanism of all MTOSI XML data based on XML Schema complex types such as the definitions of objects and notifications (events).

4.1.1 Design Guidelines

Any release of the MTOSI XSD should be customizable to fit the need of a specific solution. Note that the UML model names this extensibility placeholder to allow vendor extensions: ***additionalInfo***.

The customer extension mechanism should allow a processor to accept additional custom elements needed in a specific solution while maintaining syntactical compatibility with any processor outside the solution. In other words, a processor should be able to process the known extensions or silently ignore them if not known.

Refer to [SD0-5] for the analysis of the extensibility mechanisms of XML entities, which are based on an XML Schema complex type.

4.1.2 Usage Guidelines

All XML elements found in the ***vendorExtensions*** element of an MTOSI XML instance complex entity:

- MUST be associated with a namespace. This is a mandatory XML condition of a valid XML instance.
- Any namespaces can be used including anyone of the MTOSI XML Schema module namespaces. For example, one may re-use the XML definition of the list of name and value string pairs (***nvsList***) from the **Framework *GeneralDefinitions.xsd*** (http://www.tmforum.org/mtop/fmw/xsd/gen/v1)
- If a vendor specific namespace is used, it is recommended that its naming definition be globally distinguishable to avoid any conflict with definitions from other vendors. For example, a namespace using URN notation could use this format:
  urn:com:<company>:<context>:<version> where:
  - ***<company>*** is the name of the company that owns the vendor definition

- **<*context*>** is a unique distinguishable context within the company such as a product name
- **<*version*>** is the version of the definitions associated with the namespace

- Refer to the XML samples of managed objects, which are captured in the xml folder of the various MTOSI DDP IIS (Framework and DM DDPs).

4.2 XML Simple Type Extensions

This section describes the definitions used in the MTOSI schema to allow for extension of the MTOSI attributes of the MTOSI objects and notifications (events).

4.2.1 Patterns Overview

MTOSI identifies and supports the following XML simple type extensibility patterns:

- **Open** - the MTOSI team has not specified any specific values for this attribute. The value set is left as decision for the software vendor. Many of the name-related attributes are of this type, e.g., the name attribute for managed element.
- **Closed** - the MTOSI team has defined a discrete set of possible values for this attribute. For this type of attribute, no vendor extensions are allowed and the MTOSI team does not plan any extensions either. This is used for very stable attributes (a Boolean might be a good example).
- **Qualifiable** - the MTOSI team has determined that this attribute may need further qualification by the vendor. This means that for a given value of an attribute, the vendor may specify sub-values. Check section 4.2.3 with the example of the ***resourceState*** attribute.
- **Extendable** - the MTOSI team has determined that the value set for this attribute can be extended by a vendor (i.e., an implementer of the MTOSI) or by the MTOSI team itself (in going from one version to the next). In this case, the additional values for the attribute do not overlap in meaning with the existing values defined for the attribute. Extendable-Vendor means only extendable for the vendor, Extendable-MTOSI means only extendable by the MTOSI team within a minor version update, and Extendable means both.
- **Overlap** - the MTOSI team has determined that the value set for this attribute can be extended by a vendor. In this case, the additional values overlap partially or completely with the MTOSI provided values for the attribute. Also, the vendor must provide a mapping between the overlapping values for the attribute and the MTOSI defined values.

4.2.2 Design Guidelines

A specific XML Schema design pattern must be defined for each type of MTOSI object/notification attribute instances (value) to support their extension characteristics as described above.

The first two extension types (Open and Closed) are easily associated with an XML Schema simple type definition. The associated XML object/notification attribute instance is based on a single element text value.

The other extension types are more challenging as they can only be implemented using more complex XML Schema design patterns. The associated XML object/notification attribute instance shall rely on additional information to explicitly identify the extension type being used. And, in all these cases, the modified specification must be documented by the vendor (Refer to the SD2-1 Implementation Statement) to explain the change from the original MTOSI specification definition of the extended object/notification attribute.

4.2.2.1 Open Type Extension

Definition: Attribute is defined without any constraints.

This is only applicable to attributes that extend from the following data types; string or any numerical. Booleans are excluded as they can be associated with a simple form of enumeration.

XSD:

- ***Type***: <simpleType> any of the xsd:string or numericals datatypes.
- ***Extension***: Free as it has no constraints.
- ***Caution***: Adding to the schema some constraint settings (changing to another extension form) in the future can only be handled as a major version update! Backward compatibility would not work as the XML schema validation would be failing.

Example:

We identify that (EquipmentObjectTypeType)Equipment.expectedEquipmentObjectType is Open

```
<xsd:simpleType name ="EquipmentObjectTypeType">
        <xsd:restriction base="xsd:string">
        </xsd:restriction>
</xsd:simpleType>
```

But, changing it to the following is a Major version update (change to a Closed extension type)

```
<xsd:simpleType name ="EquipmentObjectTypeType">
        <xsd:restriction base="xsd:string">
                <xsd:minLength value="1"/>
                <xsd:maxLength value="1024"/>
        </xsd:restriction>
</xsd:simpleType>
```

Consequently, an open attribute should be restricted (specialized) by an MTOSI vendor by introducing business rules rather than restricting the XSD in order to maintain syntactical compatibility with the base MTOSI release.

4.2.2.2 Closed

Closed Attributes cannot be extended by an MTOSI vendor and will trigger a Major release if extended by an MTOSI editor in the context of a new release.

4.2.2.3 Qualifiable Type Extension

Definition: Constraint is not relevant - But, sub-definition of values shall be offered to vendor.

This is only applicable to simple type attribute.

XSD:

- ***Type***: <simpleType> any of the xsd base types.
- ***Extension***: Another attribute of the same XSD type is required to allow carrying the sub-values.
  - Naming convention: "Qualifier"
  - Constraint: Must be Open.

Example:

We identify that *(String)Common.resourceState* is Qualifiable.

The following object attribute must be available as well: *(String)qualifier*

```
<ME>
  <resourceState qualifier="IS_ACTIVE" > <!--note the extension here-->
    INSTALLED
  </resourceState>
</ME>
```

4.2.2.4 Extendable Type Extension

Definition: Schema can be extended by MTOSI team or vendor (as a minor version update)

This is applicable to the two possible types of attribute; simple and complex.

XSD:

Two scenarios based on the type of the attribute:

Simple Type Element:

- ***Type***: <simpleType> of any of the xsd base types.
- ***Extension***: It is a limitation of XML schema. Changing the constraints breaks the validation of the schema, which would normally be handled as a major version update.

  Refer to the [SD0-5] for the XML Schema extensions mechanisms analysis and the details of the option adopted in MTOSI.

Complex Type Element:

- ***Type***: <complexType> of any form.
- ***Extension***: A special ***vendorExtensions*** containment that can take any elements of any namespace can be used. Same pattern as for extension of MTOSI objects/events.

  Refer to the [SD0-5] for the XML Schema extensions mechanisms analysis and the details of the option adopted in MTOSI.

Any unplanned structural change in a complex type will trigger a major release.

4.2.2.5 Overlap Type Extension

Definition: Schema can be extended by the vendor. It is the most complex extension type as the vendor is allowed a complete re-definition of the scope and distribution of the possible values based on a mapping mechanism.

This is only applicable to simple type attribute.

XSD:

- ***Type***: <simpleType> of any of the xsd base types.
- ***Extension***: A similar pattern as the one adopted for the Qualifable extension type can be used here. Another attribute is defined to hold the vendor value that is mapped from the initial MTOSI value.

  Refer to the following convention:
  - Naming convention: "Overlap" attribute
  - Constraint: Must be Open.

Example: the same example proposed for the qualifiable extension applies replaving the attribute name “qualifier” with “overlap”.

4.2.3 Examples

We use the ***resourceState*** attribute to provide examples of the four attribute extensibility patterns; **Qualifiable**, **Extendable-Vendor**, **Extendable-MTOSI** and **Overlap**.

First, refer to the **CommonResourceInfo** XSD module in **NetworkResourceBasic** DDP IIS.

The MTOSI standard defines the simple data type definition; **ResourceStateEnumType,** which is used for the ***resourceState*** of all MTOSI managed resources has an enumeration of possible values. As is, this data type is closed for any extensions.

The complex data type definition; **ResourceStateType** is defined as simple content based on the list of standard enumerations (**ResourceStateEnumType**) with the following two additional optional attributes:

- ***overlap***
- ***qualifier***

A service provider may want to extend the standard values as follows:

1. The state "RETIRING" is further qualified as "RETIRING -REMOVED" and "RETIRING-STORED_AWAY".
2. New states "PRESENT" and "NOT_PRESENT" are used instead of "PLANNING" and "INSTALLING" but the semantics of the values are not the same. For example, let's say "PRESENT" means the equipment is in the service provider's possession and "NOT_PRESENT" has the opposite meaning.
3. A new state "SOLD" is used to indicate the equipment has been sold. For the sake of argument, assume that the meaning of "SOLD" does not overlap any of the other MTOSI-defined values for ***resourceState***.

The "RETIRING" state is extended via qualification. The ***qualifier*** XML attribute would be set to "RETIRING -REMOVED" or "RETIRING-STORED_AWAY" whenever the ***resourceState*** is set with the value "RETIRING".

The "PLANNING" and "INSTALLING" states are extended (actually overlapped) by the "PRESENT" and "NOT_PRESENT" states. In such cases, the ***resourceState*** would typically be populated with the value "UNKNOWN" and the ***overlap*** XML attribute would be set with "PRESENT" or "NOT_PRESENT" values.

The new specific vendor (proprietary) state "SOLD" is used as a value of ***resourceState***. The vendor would set the following XML entities:

- The ***resourceState*** is set with value; "VENDOR_EXT" (indicates it is an extension of the standard enumerations)
- The ***extension*** XML attribute is set with value; "PROP_SOLD" (PROP_ prefix is used to indicate this is a proprietary value)

Now if the vendor took their new state to the MTOSI team and the team approved, the set of valid states for ***resourceState*** would be expanded. This can happen in one of two ways:

1. The MTOSI team may only want to make a minor update (backward compatible). In this case the new value would be documented in a future minor version with the following usage:
   - The ***resourceState*** is set with value; "MINOR_EXT" (indicates it is an minor version extension of the standard enumerations)
   - The ***extension*** XML attribute is set with value; "SOLD"
2. On the other hand, the MTOSI team may decide to make a major update (not backward compatible). In this case, the new "SOLD" value would be added to the list of standard enumerations (**ResourceStateEnumType**).

It should be noted that the difference between the two above mentioned options is that while in the first case, there is no “type checking” on the value, i.e., the extension XML attribute value could be anything and message processor would pass it, in the second case, there is type checking and a valid value must be used or a validation exception will be raised.

4.3 MTOSI Vendor Objects and Notifications

This section describes the following other extensibility patterns, which are offered in the MTOSI Framework DDP:

- Vendor Object – Defines a proprietary object extending from the common MTOSI object
- Vendor Event – Defines a proprietary event extending from the common MTOSI event

4.3.1 MTOSI Vendor Object Specifications

4.3.1.1 Element Definitions

Refer to the Framework DDP XSD module: ***VendorObject.xsd***

This module defines a generic vendor object XML element: v***endorObject***

The Vendor Object attributes are:

- All the local XML elements inherited from the MTOSI **CommonObjectInfoType** (See Framework DDP XSD module; ***CommonObjectInfo.xsd***)
- All vendor / proprietary specific attributes shall be captured in the ***vendorExtensions*** local XML element, which is inherited from **CommonObjectInfoType**.

4.3.1.2 Naming Usage

Naming rules are as followed:

- The definition of the Vendor Object Distinguishable Name (DN) shall follow the same rules as all other MTOSI standard object names (See SD2-7).
- The Vendor Object SHALL NOT be parent of any MTOSI standard object. Meaning, the Vendor Object is always at the end of the naming tree/hierarchy.
- The Vendor Object must use a proprietary RDN that identifies:
  - The type of the Vendor Object using the usual convention for proprietary extensions: ***PROP_<Type>***, where ***<Type>*** is the vendor specific identification of the object type (class).
  - The instance of the Vendor Object is defined by a unique string value.
- A naming hierarchy of several Vendor Object types is permitted. The DN of the Vendor Object at the bottom of the tree will reflect the Vendor Object type naming hierarchy.

4.3.1.3 Notifications Usage

Notifications of a Vendor Object are supported on the basis of a Vendor Object instance name and the Vendor Object class type. For example, the ObjectCreation notification attributes are set as followed:

- ***objectName*** is the DN of the Vendor Object instance.
- ***objectType*** is the RDN used to identify the Vendor Object type.

4.3.1.4 Inventory Retrieval Usage

The retrieval of MTOSI Vendor Objects is supported using the coarse-grained operation ***getInventory***, which is offered in the **Resource Inventory Retrieval** service interface found in the **Managed Resource Inventory** DDP.

Use of the filter to query and retrieve Vendor Objects is the same as for the MTOSI standard objects.

However, all the Vendor Object instances are encapsulated in a specific list of the inventory layout data structure.

4.3.1.5 Example

See XML samples captured in the Framework DDP IIS/xml folder.

4.3.2 MTOSI Vendor Notification Specifications

4.3.2.1 Element Definitions

Refer to the Framework DDP XSD module: ***EventVendorNotification.xsd***

This module defines a generic vendor notification XML element: v***endorNotification***

The Vendor Notification attributes are:

- All the local XML elements inherited from the MTOSI CommonEventInformationType (See Framework DDP XSD module; ***CommonEventInformation.xsd***)
- All vendor / proprietary specific Vendor Notification attributes shall be captured in the ***vendorExtensions*** local XML element, which is inherited from CommonEventInformationType.
- ***type*** local XML element based on a free string. it is recommended that vendors use the following format convention: <vendorName>/<notificationType> where:
  - <vendorName> is the string uniquely identifying a vendor
  - <notificationType> is the string uniquely identifying the vendor notification type

4.3.2.2 Notification Topic Usage

This specification does not provide any association of the MTOSI Vendor Notification to one of the recommended subscription topic (See SD2-5). However, this is an aspect of the implementation that shall be covered in the Implementation Statement (See SD2-1) of any MTOSI Target OS that supports Vendor Notification.

4.3.2.3 Example

See XML sample captured in the Framework DDP IIS/xml folder: notify-VendorNotification.xml

4.4 MTOSI Extensions Usage Recommendations

The MTOSI XML Schema is designed to provide maximum extension capabilities in order to allow a complete and successful integration of various OS solutions. However, the following sections suggest some usage recommendations. These recommendations should all be understood by any implementers who are designing proprietary schema that is going to be used in these MTOSI vendor extensions.

Finally, as a general recommendation, all vendor specific definitions of interest to the MTOSI user community should be presented to the MTOSI team as potential contributions for additions to the standard interface definitions.

4.4.1 Extending an MTOSI Object vs. Using Vendor Object

Extension of an MTOSI object/notification is usually preferable to the use of an MTOSI vendor object/notification.

It is recommended to use the MTOSI object/notification extension method if the characteristics of the extension can be associated with one of the existing MTOSI objects/notifications. Extending the definition of an MTOSI object/notification with additional specific attributes has the following advantages:

- Simpler implementation effort (no specific naming handling)
- Greater management controls (retrieval, data lifecycle)
- Easier integration (the object/notification is a known entity)

For instance, if a network entity concept can be related to an MTOSI Physical Termination Point (PTP), the recommended option is to extend the schema definition of the PTP with specific additional attributes in the ***vendorExtensions*** element.

Otherwise, a specific vendor object can be defined to describe that network entity concept, which cannot be mapped to any of the MTOSI objects.

4.4.2 Extending an MTOSI Service Interface

Additional vendor operations can be added to the extend the scope of an MTOSI service interface capabilities. It is recommended that these vendor operations respect the same Web Services design guidelines used for the production of all the MTOSI IIS (Refer to the Framework DDP SD0-5).

Finally, if these vendor operations have value to the MTOSI user community, it is recommended that they be reported to the MTOSI team for evaluation and potential inclusion in a future release.

5 Advanced Versioning Features

The present MTOSI mechanism describes versioning at the MTOSI interface and data module definition level. While this granularity should address most of the concerns, the provider may implement a finer granularity. In a finer granularity scenario, an interface or even an operation may be partially compatible depending on the formal attributes or on the operation invoked in the interface. This fine grain versioning may widen the base of compatible services, allowing a service consumer to engage on a "conversation" with a mediation service checking compatibility each step in the flow. Since a fine grain versioning has a more relaxed notion of minor version compatibility, it is useful for a service consumer to modify the default behavior of the service provider to silently ignore all the unsupported tag. A way for a consumer to accomplish this is by identifying and marking the critical data with "**MUST_UNDERSTAND**" and triggering an exception if any of this data gets dropped in the service provider. The marking of the XML data can be accomplished by adding a list in the Header or by adding an attribute in each element. Given the complex nature of these extensions, the current MTOSI specifications do not support this mechanism.

6 Summary

This supporting document classifies the changes to a service interface or data module specifications in the following two categories:

- A minor change will result in a minor version increment of the associated specification modules. All minor changes are fully compatible with previous versions within the same major version.
- A major change (even if only semantic) will break the compatibility, which may require complex transformation services to allow a smooth release transition.

Various vendor extensibility patterns are supported to address any proprietary extensions a vendor may require in a particular deployment of MTOSI.

7 References

[SD0-5] – Framework DDP SD0-5_mTOPGuidelines_WebServices

[BAU04] – "Theory of Compatibility (Part 3)" – David Bau, http://davidbau.com/archives/2004/01/15/theory_of_compatibility_part_3.html

[ORC04] – "Providing Compatible Schema Evolution" - David Orchard, http://www.pacificspirit.com/Authoring/Compatibility/ProvidingCompatibleSchemaEvolution.html

[ORC01] – "Versioning XML Languages" - David Orchard, Norman Walsh, http://www.w3.org/2001/tag/doc/versioning-20031003-diff.html

[ENUM] - Managing Enumerations in W3C XML Schemas http://www.xml.com/lpt/a/2003/02/05/wxs-enum.html

[SD2-9] Using JMS as an MTOSI Transport

8 Administrative Appendix

8.1 Document History

Version Number	Date Modified	Modified by:	Description of changes
1.0	MM/DD/YY		1.0
1.1	November 2005	Included comments from ME review	1.1
2.0	April, 2008	Jérôme Magnet	Simplified/updated based on MTOSI R2 changes

8.2 Acknowledgments

First Name	Last Name	Company
Tom	Kelley	Telcordia Technologies Inc.
Josephine	Micallef	Telcordia Technologies Inc.

8.3 How to comment on this document

Comments and requests for information must be in written form and addressed to the contact identified below:

Francesco	Caruso	Telcordia Technologies Inc.
Phone:	+1 732 699 3072	
Fax:	+1 732 336 7026	
e-mail:	caruso@research.telcordia.com	

Jérôme	Magnet	Nortel

Phone:	1-613-763-1480
Fax:	
e-mail:	jeromem@nortel.com

Please be specific, since your comments will be dealt with by the team evaluating numerous inputs and trying to produce a single text. Thus we appreciate significant specific input. We are looking for more input than wordsmith" items, however editing and structural help are greatly appreciated where better clarity is the result.