EBU

OPERATING EUROVISION AND EURORADIO

TECH 3356

SPECIFICATION OF THE FIMS MEDIA SOA FRAMEWORK

GENERAL DESCRIPTION

VERSION 1.3.1

Published Jointly With FIMS and AMWA

Geneva
October 2017

[*] Page intentionally left blank. This document is paginated for two sided printing

Executive Summary

This document describes a vendor-neutral common framework for implementing Interoperable Media Services using a Service Oriented Architecture (SOA) based system, supporting interoperability, interchangeability and reusability of media specific services.

The FIMS 1.2 release at the time of publication of this document comprises the following:

- FIMS 1.2 General Description (this document):
  FIMS Schema Spec-1.2.pdf
- FIMS API documentation
  APIDocumentation/START HERE.html
- FIMS schema:
  In folder *WSDL-SOAP-XSD*

NOTES – The user's attention is called to the possibility that implementation and compliance with this specification may require use of subject matter covered by patent rights. By publication of this specification, no position is taken with respect to the existence or validity of any claim or of any patent rights in connection therewith. The AMWA, including the AMWA Board of Directors, shall not be responsible for identifying patents for which a license may be required by an AMWA specification or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

[*] Page intentionally left blank. This document is paginated for two sided printing

Contents

Framework for Interoperable Media Services
General Description

EBU Committee	*First Issued*	*Revised*	*Re-issued*
TC	2015	Oct. 2017	

Keywords: FIMS, AMWA, SOA, Service Oriented Architecture, Media Services, SOAP, REST

1. Scope

This document describes a vendor-neutral common framework for implementing Interoperable Media Services using a Service Oriented Architecture (SOA) based system for use in broadcast, production, post production, media distribution, and media archive applications. The framework supports interoperability, interchangeability and reusability of media specific services.

The high-level architecture and framework is described. This framework covers the following system and management requirements: service management, awareness, behaviour and communication, content and time awareness.

This first version of the specification addresses the following basic Media Services: Capture, Transform, Transfer, Quality Assessment, Repository management and Automatic Metadata Extraction.

This document should be read in conjunction with the following:

- FIMS schema files comprising WSDL and XSD definitions
- FIMS API documentation

The standard major, minor and revision numerical versioning system is used for release management of the specification. 1.3.x releases of the specification shall contain minor revisions addressing inconsistencies and defects within the basic 1.3 release. Additional features to the 1.3 release will only be added to a future 1.4 document.

2. Conformance Language

Normative text is text that describes elements of the design that are indispensable or contains the conformance language keywords: "shall", "should", or "may". Informative text is text that is potentially helpful to the user, but not indispensable, and can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except: the Introduction, any Section explicitly labelled as "Informative" or individual paragraphs that start with "Note:"

The keywords "shall" and "shall not" indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted.

The keywords, "should" and "should not" indicate that, among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

The keywords "may" and "need not" indicate courses of action permissible within the limits of the document.

The keyword “reserved” indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword “forbidden” indicates “reserved” and in addition indicates that the provision will never be defined in the future.

A conformant implementation according to this document is one that includes all mandatory provisions ("shall") and, if implemented, all recommended provisions ("should") as described. A conformant implementation need not implement provisions ("may") and need not implement them as described.

Unless otherwise specified, the order of precedence of the types of normative information in this document shall be as follows: Normative prose shall be the authoritative definition; Tables shall be next; followed by formal languages; then figures; and then any other language forms.

3. Reference Documents

3.1 FIMS

All the FIMS documentation and schema are on https://github.com/fims-tv/fims

1) FIMS 1.3.1 schema files in folder *WSDL-REST-XSD*

3.2 General References

2) World Wide Web Consortium (W3C), http://www.w3.org

3) Organization for the Advancement of Structured Information Standards (OASIS), http://www.oasis-open.org

4) Web Services Interoperability Organisation (WS-I), http://www.ws-i.org

5) Advanced Media Workflow Association (AMWA), http://www.amwa.tv

6) Society of Motion Picture and Television Engineers (SMPTE), http://www.smpte.org

7) European Broadcast Union, EBU Technical References, http://tech.ebu.ch

3.3 Standards References

Implementers should have a general understanding of the following technologies:

3.3.1 Web and Internet Technology

8) RFC 1738: Uniform Resource Locators (URL)

9) RFC 1521: Mechanisms for Specifying and Describing the Format of Internet Message Bodies

3.3.2 SOAP Web Services

10) WSDL 1.1

11) Web Services Description Language (WSDL) Version 2.0 Part 0: Primer - W3C Recommendation 26 June 2007. Available at http://www.w3.org/TR/wsdl20-primer.

3.3.3 Date and time formats

12) EBU Tech 3295 P_META Metadata Library

13) ISO 8601:2004 Data elements and interchange formats Information interchange Representation of dates and times

14) IETF RFC 3339 Date and Time on the Internet: Timestamps

3.3.4 EBU studies/specifications and SMPTE Standards

15) EBU Tech 3293: EBU Core Metadata Set (EBUCore) – latest version

16) EBU Tech 3295: P_META Metadata Library

17) SMPTE ST 12M-1:2008 Television – Time and Control Code

18) SMPTE RP 224v11:2011: SMPTE Labels Register

19) SMPTE ST 258:2004 Television - Transfer of Edit Decision Lists

20) SMPTE ST 291:2010 Television - Ancillary Data Packet and Space Formatting

21) SMPTE ST 377-1:2009 Material Exchange Format (MXF) – File Format Specification

22) SMPTE ST 330:2004 Television - Unique Material Identifier (UMID)

23) SMPTE ST 434: 2006 Material Exchange Format - XML Encoding for Metadata and File Structure Information

24) SMPTE ST 436: 2006 Television - MXF Mappings for VBI Lines and Ancillary Data Packets

25) SMPTE ST 2071-1:2014 Media Dispatch Protocol (MDP) - Protocol Specification

4. Overview

This Section is informative.

This specification defines a common approach to the integration of software components in modern media production facilities, which is believed to be a fundamental need of the entire media industry.

The specification is based on an overall framework for integration of reusable components for multimedia content production, which would support the business functions of the professional media industry. This framework for media services includes specific detailed definitions of common media service interfaces.

The focus of this specification is on Service Oriented Architecture (SOA), and reflects a move by the media industry toward systems that use this approach. Media companies are moving towards more rapid adoption of these systems because of a need to increase agility in a market where user requirements are changing rapidly. In addition to the need for agility, these companies require the increased maintainability, scalability and extensibility that standardized service interfaces provide.

To properly exploit this technology a common framework should be adopted to help ensure integration interoperability, interchangeability and reusability of services. This will drastically reduce integration costs, allow users to more freely choose the most appropriate products on the market at any given time, improve maintainability, and aid in the adoption of new technologies.

Section 5 gives an overview of the high-level architecture, and provides the overall reference model of the FIMS Framework.

Section 6 describes media service management, and specifies Service lifecycle and the lifecycle of a job.

Section 7 describes media service awareness, including the Service registry, and Service description.

Media Service behaviour is described in **Section 8**. This includes the service behaviour that is common to all categories of services. Behaviour specific to each service category is also described.

Details of the data model and communication protocol of FIMS are specified and described in the self-documenting technical schemas and related artefacts in the WSDL-REST-XSD folder. Details on a REST implementation of FIMS are given in the documentation of each service.

Future versions of the FIMS specification will continue to expand the number of supported media service interfaces.

5. High-Level Architectures

5.1 What does it mean to be 'FIMS Compliant'?

Compliance with FIMS requires that the following conditions shall be met:

1) Messages shall be implemented as described in the FIMS service description framework

2) Message names or defined schemas shall not be modified.

3) Messages shall not be processed in a means that is dependent on schema extensions.

4) In the case of SOAP implementations, each service interface shall comply with the FIMS WSDL. For REST implementations, messages shall comply with the header and body REST mappings defined by the specification for each service, Message payloads shall validate with the FIMS schema.

5) In the case of SOAP implementations, each service shall implement at least the mandatory parts of the FIMS WSDL definitions. Similarly, REST implementations of services shall implement at least the mandatory parts defined in the common FIMS schemas. Definitions shall not be extended except for fields that explicitly allow vendor extensions.

5.2 Reference architecture(s)

Figure 1 shows the overall reference model of the FIMS Framework, including areas not specified in FIMS.

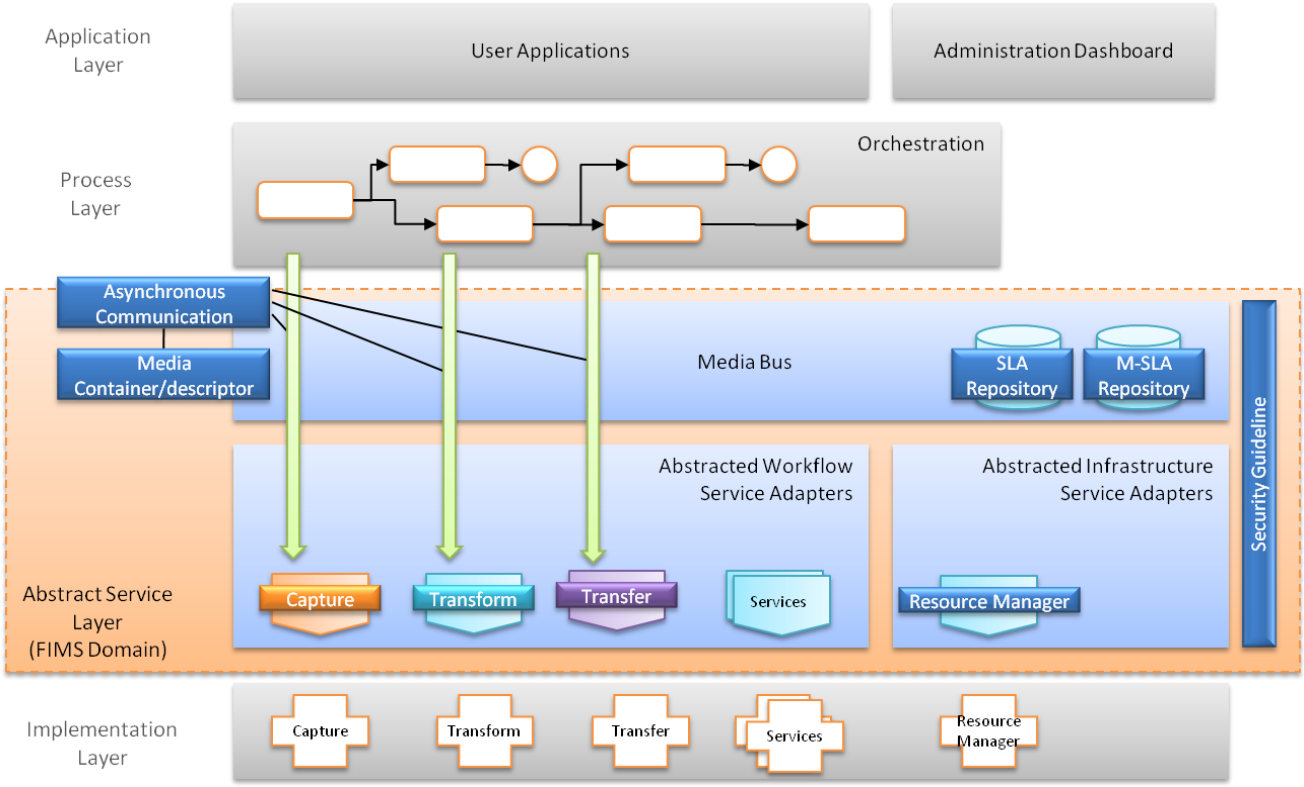

Figure 1: FIMS Framework reference model

To take account of the specific needs of audiovisual media, the FIMS framework has to consider aspects of media services that may differ from conventional IT-based SOA such as:

1) Asynchronous communication to properly handle the very large amounts of data associated with AV media in a timely manner.

2) Media Container/Media Descriptor to associate AV metadata with AV essence.

3) Media Infrastructure Service (Resource Manager) for appropriate media handling.

4) Media Bus for AV data exchange, in addition to the conventional ESB (Enterprise Service Bus) for message exchange.

5) Media Bus M-SLA (Media-Service Level Agreement), in addition to the conventional SLA in ESB.

6) Security guidelines for secure media handling.

Media-centric issues such as latency processing big files are described in Section 5.5.

Figure 2 illustrates the scope of this version of the FIMS Framework, i.e. the definition of service adapters.

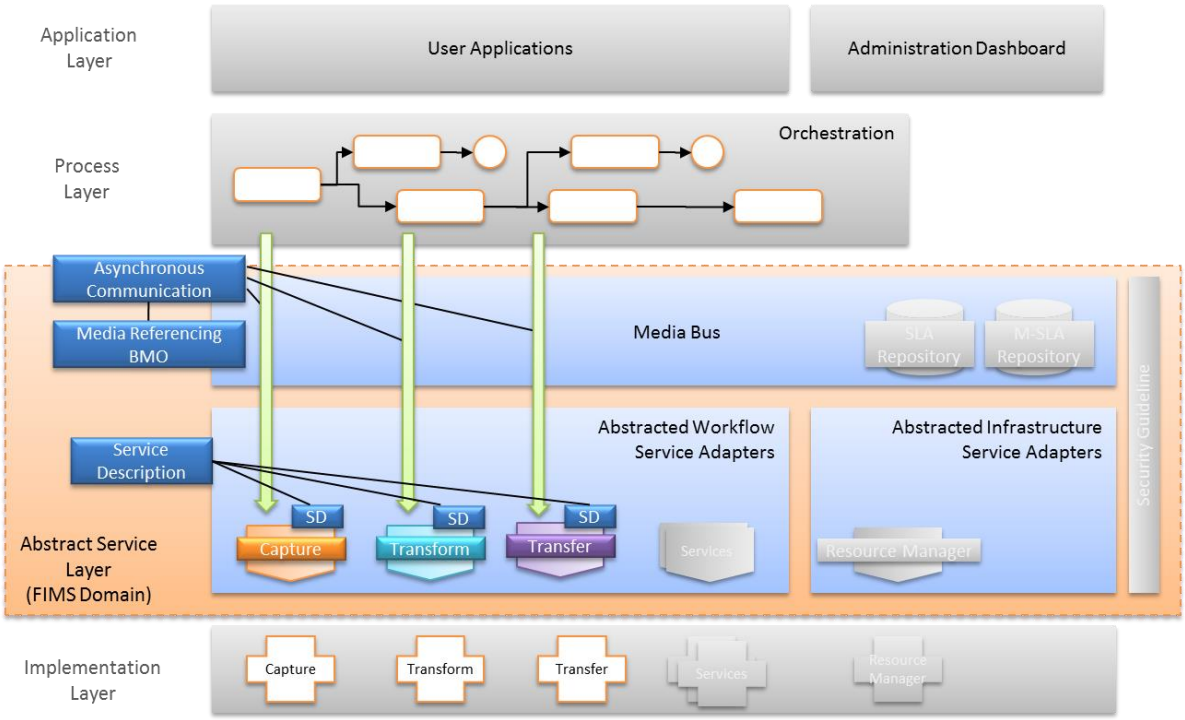

Figure 2: FIMS framework

The reference architecture is capable of working with different SOA architectural approaches, supporting SOAP/WSDL and RESTful interaction styles.

5.3 Service Taxonomy

A system with FIMS components contains two broad service categories: (1) workflow services able to realize a given business goal (Media Workflow services), (2) infrastructure services that are essential components of the Media SOA system (Media Infrastructure services).

Infrastructure services that are used to construct the Media SOA system are called Media Infrastructure (MI) services. The MI services would in the future include a Resource Manager that conducts the resource allocation, as well as other common services like job scheduling and queuing (e.g. using AWS step functions).

Depending on the type of business workflow, the number and type of media services used may vary.

5.4 Scenarios: Mediations, Dynamic resource allocation

The FIMS framework has been conceived in such a way as to allow the implementation of systems with varying levels of complexity, depending on the objectives, operational environment and user expectations.

At the simplest level, services can be invoked directly by consumers (user interface clients or orchestrators), without any intermediate software layer. In this case the workflow orchestrator takes all the responsibility for any supporting operation such as service discovery, resource balancing, etc. In small to medium environments this may be appropriate. In more complex

installations, where several processes must run in parallel and compete for resources, it may be convenient to centralize the management operations in an intermediate software layer. This is usually referred to as the Enterprise Service Bus (ESB).

Features that can be associated with an ESB include:

- centralised enforcement of security;
- dynamic resource allocation;
- service registry management for capability and availability discovery;
- load balancing;
- fault tolerance;
- mediation operations, such as protocol translation (for example, from SOAP to REST).

The definition of functionalities that such an intermediate layer should have is outside of the scope of this version of the specification.

5.5 Media Centric Issues

5.5.1 Asynchronous operations for long running process

SOA-based media workflows are often long running processes, sometimes active for hours, days, or even weeks. This places specific persistence requirements of the SOA BPM platform. Servers may be stopped or restarted while processes are running. The system needs to be able to restart at the same point in the workflow and process orchestration without loss of state or data.

Many SOA-based media services run on external hardware or are software-based systems that operate in a loosely coupled asynchronous environment or can be accessed in the cloud. If the orchestration system is stopped, these services may continue running, and the job process states have to be recovered after the system restarts.

Asynchronous operation status updates may be implemented with either callbacks or polling depending on the SOA platform architecture and service capabilities.

The details of asynchronous communication are discussed in Section 8.

5.5.2 Process Scheduling and Resource Management

Because of long processing time resulting from the huge size of AV data, process scheduling and resource management can be crucial for SOA-based media systems.

Although the concept of the Resource Manager Service (see Figure 3) includes process scheduling and resource management, process scheduling may be invoked by only the orchestration system. The Resource Management service may be invoked by the orchestration system, by the mediator in the middleware, or by any of the workflow services.

The interface for a Process scheduling service could be hidden if it is only invoked within the closed orchestration system because it is left to the orchestration system developer. However, if it is also to be used by the workflow services, the Resource Manager interface could be defined as a custom FIMS-like service adapter, depending on the environment.

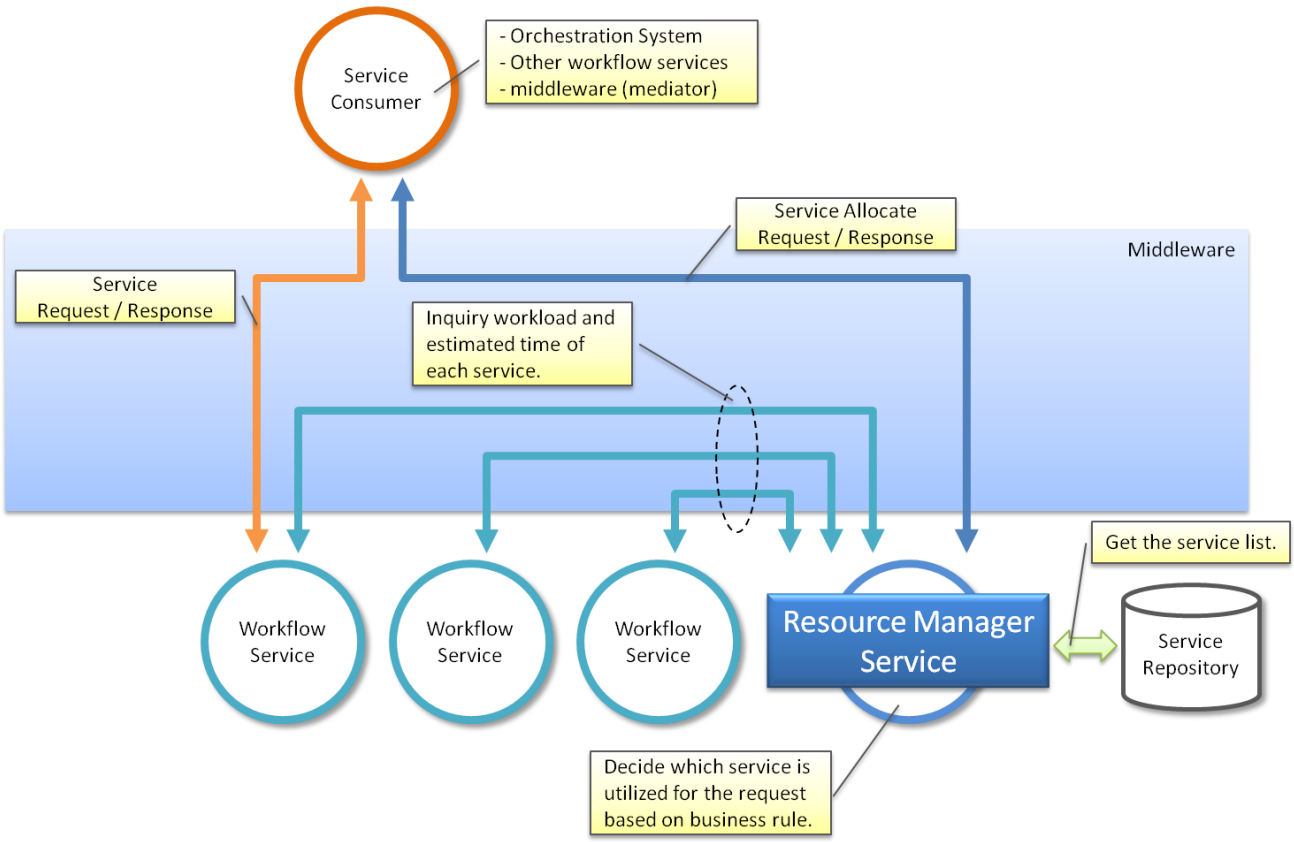

Figure 3 Resource Manager Concept

5.5.3 Media Bus

An extension to SOA often called the "Media Bus" can facilitate storage and media file centric operations. The Media Bus extension is similar to an Enterprise Service Bus (ESB) optimized for large files.

Since the system manages the storage and movement of a very large number of large files (sometimes millions of files per project) it is important that an asset data repository be available as part of the system core functionality, or as an external (third party) service.

There can be many copies of file instances and also many versions located in multiple storage areas. There may also be many lower resolution video proxy files in the system representing the original high resolution "camera negative" files or file sequences. It is necessary to be able to keep track of all these proxy files, including the different versions that may exist.

The management of files is left to the integrator according to its implementation environment and in-house requirements or structural capabilities.

5.5.3.1 M-SLA

The performance of the individual components of the Media Bus (mainly storage devices) such as bandwidth, transfer speed, capacity, may need to be queried using a common message format.

Based on this, a client such as the orchestration system and/or the media infrastructure service can select the storage device that is best suited to the client, and/or more precisely estimate the execution time to be requested.

This corresponds to SLA in a conventional IT-based SOA system and is referred to as M-SLA.

5.5.4 Security

Due to the high value of the intellectual property passing through media systems, it is critical that security be maintained and access provided only to those with proper authorization. There are several types of security that can be implemented across the SOA: agent-based security, message based security, watermarking, and Digital Rights Management. Typical media enterprises will require most if not all of these security provisions. Agent based security involves keeping track of the various participants in the SOA-based media system, and doing this correctly will no doubt require some sort of identity management infrastructure.

Identity management technology is well developed in IT, and can be put to very good use in SOA middleware. Instead of using disparate repositories and application-specific methods to authenticate users and secure systems, identity management tools allow the integrator to unify all of an enterprise under a single repository and management system of user data. This allows easy changes to user information and quick provisioning of new users. In an integrated SOA, identity management solutions also allow for role-based views into data.

FIMS service adapters have been designed to allow extensions for ad-hoc security.

6. Media Service Management

6.1 Service lifecycle

The main lifecycle states of a service are:

- Service deployed
- Service updated
- Service replaced
- Service decommissioned

6.1.1 Deployment

A service is deployed when the new service or service instance is registered in the service registry. The information provided in the registry includes:

- Service interfaces.
- Service endpoint information.
- Service description metadata.
- Service policies.

6.1.2 Decommissioning

Decommissioning services from the environment is supported by the service registry, which allows the system administrators to first deprecate existing implementations so that potential new consumers do not see the specific implementation. Administrators can then use reporting and impact analysis capabilities in the service registry to allow the operations team to identify remaining service version consumers and ensure that they migrate onto the correct alternative version. After all of the consumers have been migrated, the old service version can be retired from use and removed from the environment.

6.1.3 Replacement/Upgrade

A service can be updated or replaced by updating the service information in the service registry.

The service interfaces and schemas as specified in this FIMS specification shall not require updating if a service is replaced by one providing the same business functionality with the same version of the interface. However, as FIMS specifications evolve, new versions of an interface may be published and may need to be updated in the registry. As service versions are superseded by new implementations which deliver the same required business capability, the service governance lifecycle should allow older versions to be deprecated and, ultimately, retired.

6.1.4 FIMS Interface Versioning

Backward compatibility issues might arise for a variety of reasons. This version of the specification focuses on two scenarios:

- There is a new version of the FIMS specifications for a service interface or base schemas.
- A service implementation has been modified for adding or removing features (such as operations).

6.1.4.1 New version of interfaces

From time to time, the FIMS technical board will issue new versions of the FIMS normative schemas and associated specifications with a common, incremented version number. Although every effort will be made to make new minor version updated to be backwards compatible with previous versions, the nature of FIMS as a bi-directional communication framework causes unavoidable issues for implementers, including:

- Mismatched versions of data between that sent to a service and received in a synchronous response;
- Similarly for notifications sent from services;
- Use of runtime validation and/or auto-generated code that is specific to a schema version.

Services and orchestration systems are not required to provide simultaneous support for multiple versions of FIMS. A single service endpoint shall provide support for one and only one version of the FIMS schemas and specifications. A service may support more than one version of the FIMS specification through the provisioning of additional endpoints, one per version.

The FIMS base services schema contains a simple type called "CurrentVersion" that shall be a string value restricted to contain a single value equal to the version number of the schema. For example, the fixed value is "v1_3_0" for FIMS version 1.3.

With the exception of FIMS v1.0.7 messages, all FIMS SOAP request messages, response messages, notification messages and event messages shall indicate their version number using a mandatory attribute named "version" of "CurrentVersion" type. A response message or notification message generated by a service endpoint shall have the same version number as that of the endpoint, thereby ensuring that servers and clients support the same version.

As the version attribute was not present in FIMS version 1.0.7, any SOAP request, response or notification message that does not contain a version attribute shall be treated as a FIMS 1.0.7 message.

All FIMS REST messages other than fault messages shall contain a mandatory header named "X-FIMS-Version" with a value set to the same as the version number of the specification as the fixed value of the "CurrentVersion" simple type.

Note: No FIMS REST mapping was defined for FIMS 1.0.7, so the legacy issue of a missing version attribute is not an issue for REST services.

So that both REST and SOAP services may receive version-related fault messages, fault messages

shall not have a version number.

When the version number of a message matches that supported by the receiving endpoint, the message shall be accepted for processing.

For REST services, if the version number of a message is not supported by the receiving endpoint, it shall be rejected with a fault response with code "SVC_S00_0019", indicating a version mismatch. To enable the consumer of the endpoint to change the version of their request or route the message to an appropriate endpoint, the fault's message should contain details of the current version supported by the endpoint.

For SOAP services, if it is not possible to differentiate a version mismatch from an invalid XML instance of an xml schema, it is advised that all fault responses with code "DAT_S00_0001" should explicitly state the current version of the service in either the *description* or the *detail* part of the fault message.

Note: It is anticipated that translation endpoint services can be implemented to help users with the process of migration from one version to the next.

6.1.4.2 New service implementation

In this scenario there is no change in the version of the interface, but the service is replaced with a different implementation in which either only some of the features (e.g. operations) are implemented or additional new features are implemented. In the latter case, the service is backward compatible. In the former case, there is the issue of making the framework aware that the service does not implement some features.

The following practices shall be followed:

- Service description for each service implementation provides a list of available service operations and properties (e.g., a transform operation provides a list of supported formats).
- A new service may be invoked through mediation in the ESB or via an orchestration system, in which case a lookup in the service registry can be used to find the service instance(s) that implement the requested operation.

6.2 Job management

6.2.1 Lifecycle of a job

Figure 4 shows the states associated to a long-running job since its request until its completion, cancellation or failure. It also shows the job commands or actions that initiates a transition to a new state.

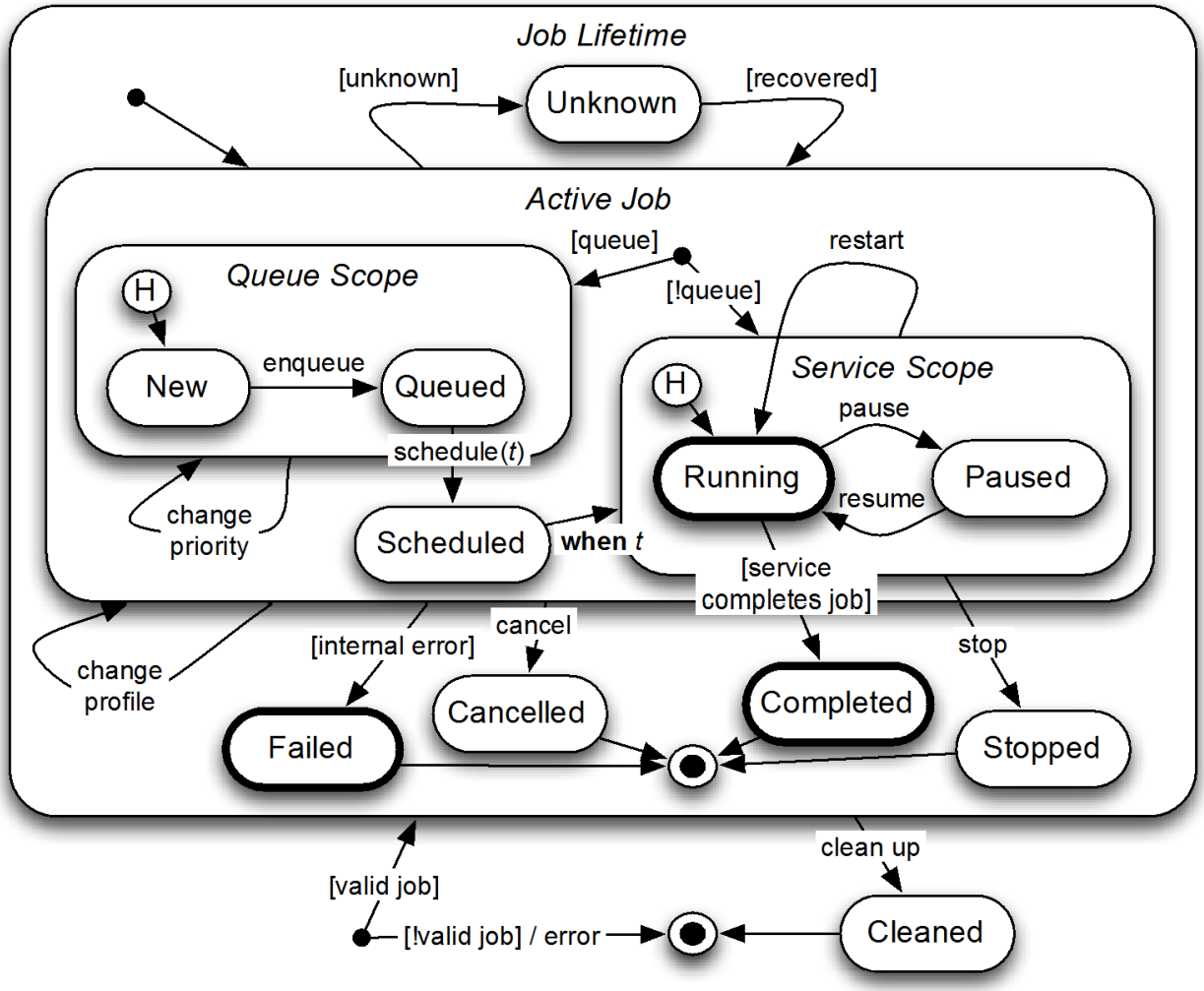

Figure 4: UML statechart for a long-running job

The thick circles in Figure 4 indicate the mandatory states that shall be implemented.

When a job request is received by a service its initial state is either *New* or *Running*. In the *New* or *Queued* state, a job may have its priority changed before starting its execution through external job management commands (if the service supports these features). The job may also be cancelled by an external command. Once the job is de-queued for execution it moves to the *Scheduled* state and when the scheduled time comes, it moves to the *Running* state. In case the service does not implement the job queue, the job goes directly to the *Running* state upon the arrival of a request.

The *Running* state indicates a job is being executed. An external command may pause the job (and resume it later). The request may also be restarted or cancelled by external commands. Restarting means to start its execution again. If a job executes until its completion it transitions to the *Completed* state. If an error occurs during its execution it moves to the *Failed* state. An external command may force the early termination of a job execution. To stop a job means to force its completion. This is not an error situation and the result of the job processing until that moment is considered to be the result of the job execution.

Media services often produce large media files that need to be available until the client or other services retrieve them. Once the job completes (or it is stopped) the resulting product of the operation shall be kept by the service. An external (or internal) command (cleanup) is used to indicate to the service that the result is no longer needed and the job transitions then to *Cleaned* state. Services may independently move jobs to the *Cleaned* state after an elapsed time, the duration of which is to be determined by the service. Services shall be responsible for cleaning jobs across service stop or crash events.

The transition to *Cancelled*, *Completed*, *Stopped* and *Failed* states shall produce a notification to the endpoints specified at the *notifyAt* parameter of the request message, if a notification is expected at the end of the job execution or at the job cancellation. *Completed*, *Stopped* and *Cancelled* shall produce a response notification message to the replyTo endpoint, while Failed shall produce an error notification message addressed to the f*aultTo* endpoint.

The *Unknown* state indicates the job is currently being processed but its state cannot be obtained.

6.2.2 Management Commands

The FIMS service interfaces provide three job management operations. They are the *manageJob*, *manageQueue* and *queryJob* operations, as shown in Figure 5.

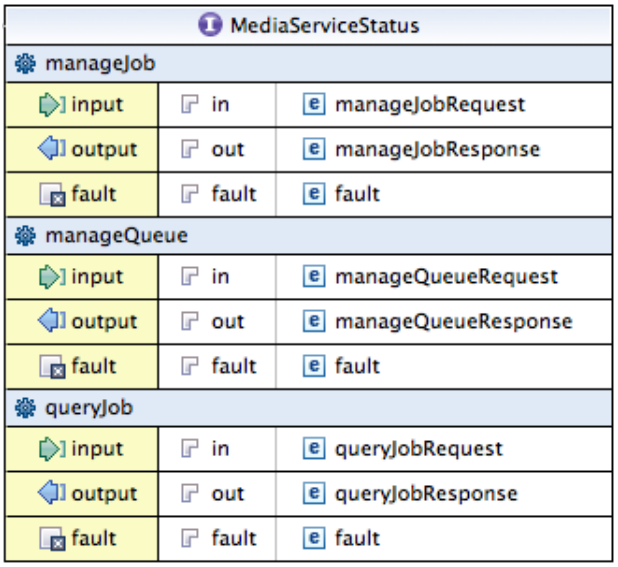

Figure 5: Job management operations

The *manageJob* operation allows a requester to send job commands to change the state of a job as described in the previous section.

If the service implements a job queue, this queue can be managed using the *manageQueue* operation. The queue commands and the associated state transitions are described in Section 6.2.1.

The *queryJob* operation returns information about jobs that were submitted to the service. The requester may provide a list with the ID of the jobs it wants the information on.

Alternatively, it can request information of jobs that meet requirements specified by a filter object.

The response list contains the identification of the jobs and detailed information about each job such as *estimatedCompletionDuration* and *status*.

6.2.3 Resource-oriented data model

Messages exchanged about jobs, the services they perform, their profiles, queues and the objects that they operate on conform to a resource-oriented data model. The classes of this data model and their relationships are shown in the UML class diagram in Figure 6. The non-referential properties of each class are not shown.

Of course profiles and jobs apply to new FIMS services not represented in Figure 6.

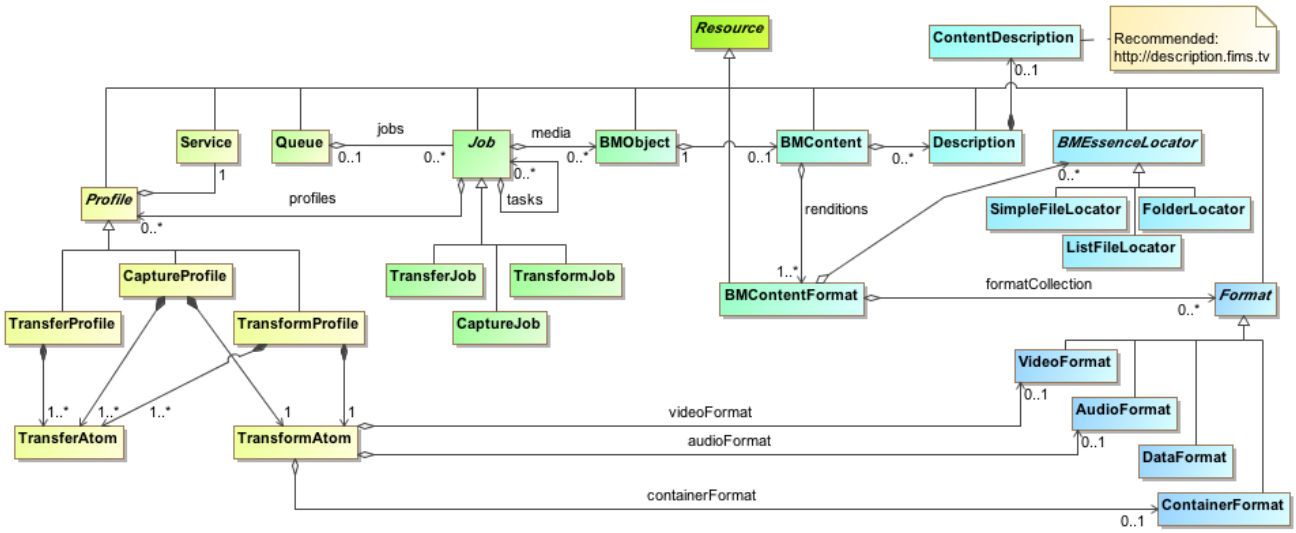

Figure 6: FIMS resources data model

Each resource has a unique identifier property "resourceID" that can be used to make reference to it, just like a hyperlink. In combination with a means to resolve references, a resource-oriented approach enables the following:

- reference can be made to shared and repeated resources so that smaller messages can be exchanged. For example, a common profile can be defined for a regularly repeated transform operation and stored in a central location. It does not have to be repeated in every job request, response or status query.
- the location of a resource can be separated from its use, facilitating geographic scalability and greater resilience through the use of resource-resolution technologies such as DNS.
- support for the introduction of data authority services that provide a single point of authority for classes of metadata. For example, a digital asset management system is used as the data authority for information about content. Rather than a copy of the information about the content being kept by each of the systems involved in a job, with all the associated overhead of keeping a copy, reference can be made to the authoritative version in the asset management system.
- a greater degree of loose-coupling, including more agility to introduce loosely-coupled monitoring systems and dashboards.
- RESTful bindings for resources.

An optional "revisionID" property may be used to keep track of revision numbers for a resource, supporting systems that use an eventually consistent approach.

An optional URI "location" property may be used to provide a specific location for the resource (URL) that can be referenced without the need for resource identifier resolution.

In terms of the normative representation of the data model as an XML schema, all properties for resources are marked as optional from an XML perspective (minOccurs="0"), with the exception of the "resourceID" property. This mechanism allows a resource to be included by reference rather than embedding it.

To encode a reference to a resource rather than embedding it, omit all of its properties other than "resourceID", "revisionID" (where used) and "location" (where used). The decoder of the message is then expected to either resolve the reference to the resource or report a fault.

To encode a resource by embedding its value, include the value of at least one property other than "resourceID", "revisionID" and "location". In general, all known property values for a resource are

encoded to minimize the requirement to merge versions of the resource. The decoder is not expected to resolve the resource by reference externally and may update an internal copy of the resource based on the information provided.

Properties that are defined as mandatory through specification, in either a request or a response message and in at least one job state, must be supported by all FIMS implementations.

Note: Mandatory properties are in request or response messages are defined per property in the XML schema annotations with source "urn:x-fims:inclusionInRequest" and "urn:x-fims:inclusionInResponse".

FIMS defines both normative technical metadata (e.g., Format, VideoFormat, AudioFormat) and recommended descriptive metadata (ContentDescription). Users are encouraged to use the core descriptive metadata provided in the FIMS schema to improve interoperability. Both technical and descriptive metadata are based on EBUCore, an EBU extension of the Dublin Core for media.

7. Media Service Awareness

7.1 Service registry

A registry is useful for tracking deployed services throughout an infrastructure. By making this registry machine-readable and well defined the overall system can make intelligent decisions.

7.1.1 Listing registered services

A FIMS system may provide a registry, in which case it shall contain a line-delimited list of URLs. This list shall be made available via an HTTP GET.

Each URL within the list shall return a response compliant to the service description document defined in Section 7.2.

The URL within the registration file may provide a hint as a 'Service' query string indicating the Service Description that will be discovered when the URL is queried.

For example, the registry could contain URLs such as the following; including those that assist in the discovery of, say, a Capture service:

http://some.dns.entry/
http://some.dns.entry/AnyURL
http://192.168.1.1:8888/SomeURL?Service=Capture
http://192.168.1.1:8888/SomeOtherURL?Service=Transform

A single description discovery URL shall only describe a single service endpoint and a single capability.

For example, the following is invalid:
http://192.168.1.1:1234/YetAnotherURL?Service=Transform,Capture

7.2 Service description

Services should support a mechanism for publishing their capabilities and configuration to systems (and humans). Without understanding what a service will accept or can process, orchestration choices become constrained and out-of-band decisions are required. Whenever information is moved out-of-band, the ability of a system to automatically react is compromised. Either the

system must wait for human intervention, or the information must be brought back into the scope of the system via extension.

For this reason, the FIMS specification defines a machine-readable model for providing information about operations a service will accept along with the current configuration of the service. This model references EBUCore Metadata.

The service registry (Section 7.1.1) defines URL locations that shall be used to return a service description XML document. When requested, a service shall return a document describing the capabilities and configuration of that service. Such attributes of a service may change over time, and the document returned may vary during the lifetime of a deployed service, for example, due to system faults, licensing restrictions, software upgrades or system administration restrictions to operations.

Note: As of FIMS v1.2, the service description may be structured as a Media Device and Control Framework service capability description. See the definition of "serviceInformation" as part of the "ServiceType" resource.

A single service instance may implement multiple classes of operation (such as a service that can perform transformation as well as plain transfer services) the service shall not register these on a single endpoint. Such a service shall register multiple endpoints.

8. Media Service Behaviour

This Section addresses the service behaviour that is common to all categories of services. Behaviour specific to each service category is also described.

8.1 Common Service Behaviour

8.1.1 Resource-oriented dialogue

Communication in FIMS consists of a dialogue of messages about the FIMS-defined resources (job, queue, service, profile, BMObject etc.) between a service and its client. An operation of a media service is executed through a dialogue about a job resource between the service provider and the operation requester.

FIMS defines a set of well-known operation implementation patterns that are supported through the SOAP/WSDL and RESTful service definitions provided, as described in Section 8.1.2. The corresponding messages embed representations of resources and/or resource references by identifiers as parameters to operations. The WSDL and RESTful service definitions provide for synchronous and asynchronous requests, responses, faults and notifications.

The RESTful approach to FIMS uses standard HTTP verbs with URI paths to achieve the same dialogue, with the resource description embedded directly as the message body. The specification of specific bindings between the HTTP verbs, headers, status codes, the FIMS-defined operation patterns and a RESTful event mechanism for notifications is provided with this version of the FIMS specification.

Note that one service provider could simultaneously support both SOAP/WSDL and RESTful interaction styles for the same resources.

Figure 7 shows an example sequence diagram of Capture Request/Ack using the SOAP approach.

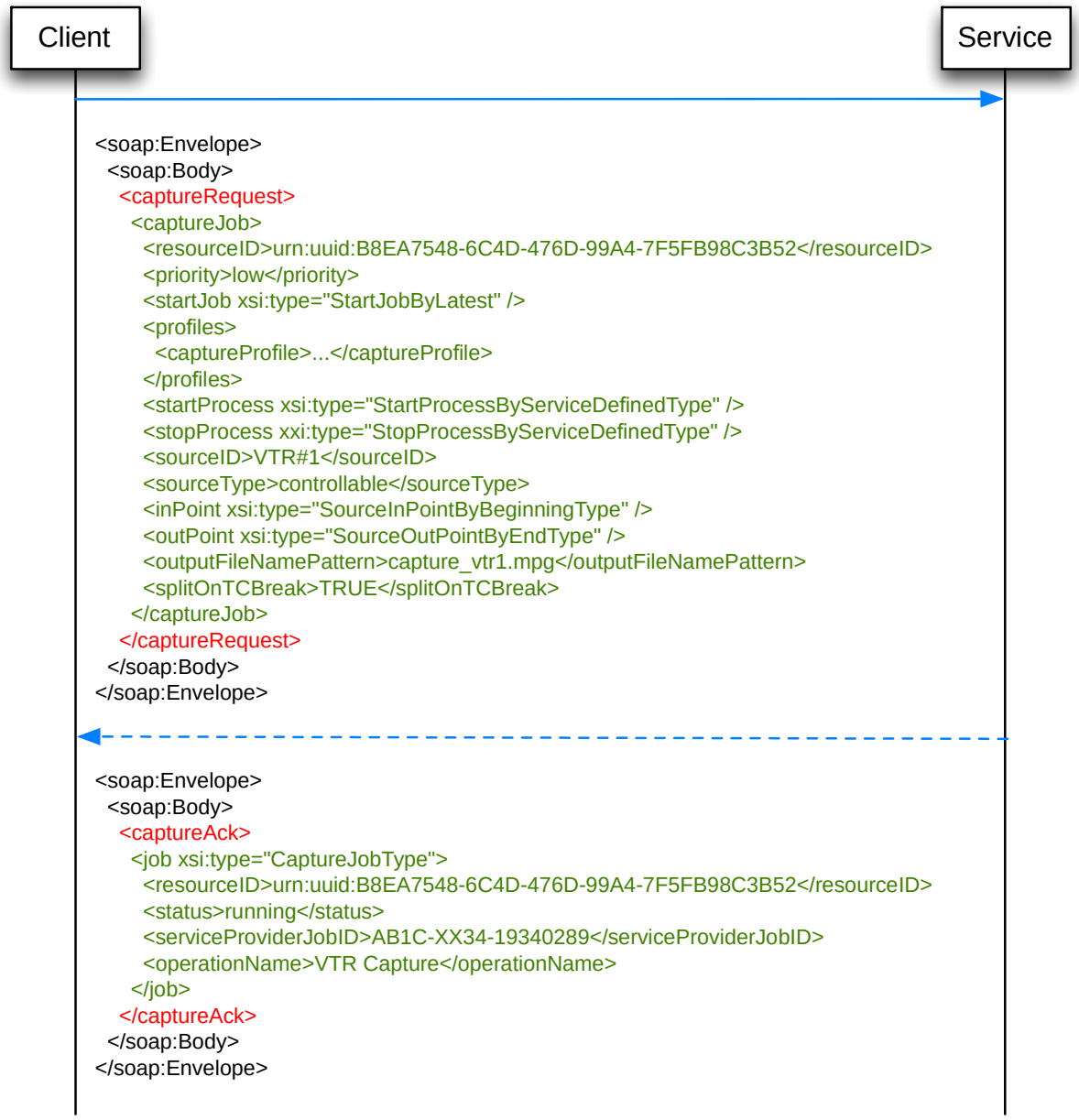

Figure 7: Service Request/Ack (Capture) - SOAP-RPC

The RESTful equivalent carries the part of the Capture Request message coloured in green as the body of a message POSTed to a service. If successful, the synchronous HTTP response has a status code of 201 Created with a body similar to the green part of the Capture Ack message.

The content type of a RESTful FIMS message body shall be represented in either XML or JSON format. Definition of the mapping of the FIMS XML schema definitions to JSON format messages is provided in Section 8.3.

The body of an example JSON Capture Request message is shown in Figure 8.

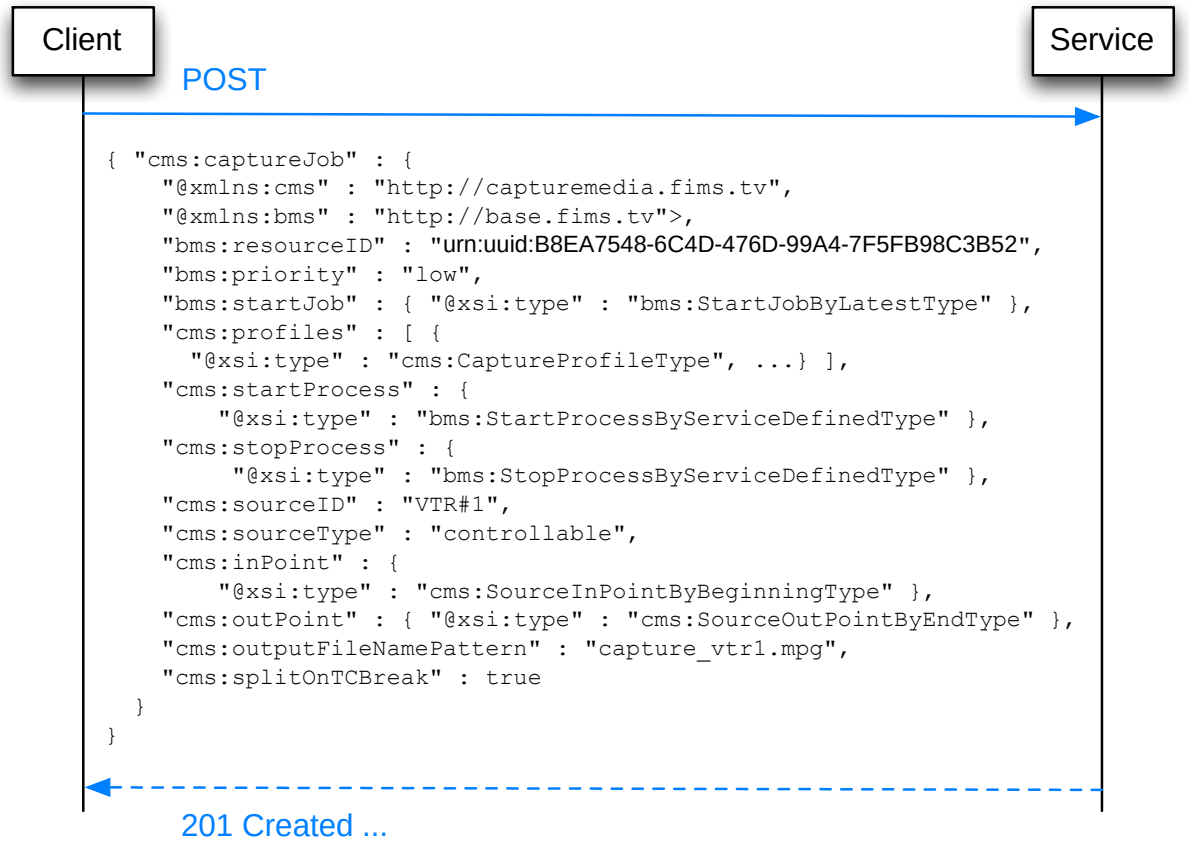

Figure 8: Service Request (Capture) - REST JSON

8.1.2 Operation Implementation Patterns

Service operations defined by FIMS interfaces provide different types of interactions between the service provider and the service requester. Each one specifies how the result of an operation is made available to the requester. The interface definition along with the input parameters of the operation determines how the service should return the response of the operation.

8.1.2.1 Synchronous Request/Response

In this interaction mode the service client (e.g. a business process) invokes the service to perform an operation passing the input parameters (*par1*, … *parN*) and receives the response in the same communication session as the request. Operations that implement this mode of interaction should not be long-running processes to avoid blocking the service client for a long period of time and to prevent timeouts that may occur in the communication session. See Figure 9.

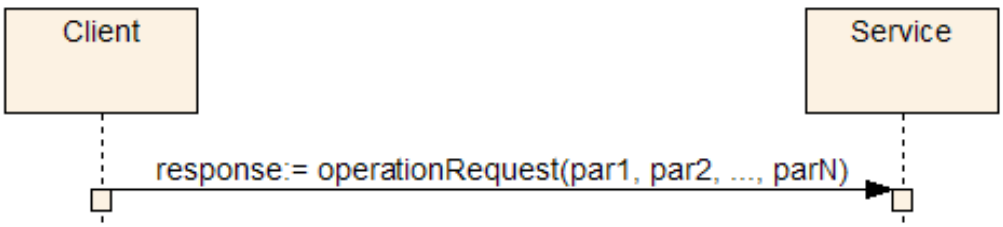

Figure 9: Synchronous request/response

Examples of operations that use this type of interaction are the job and queue management operations.

8.1.2.2 Asynchronous Request/Response with Notification

This interaction pattern should be used for long running processes. The request and response associated to the operation are exchanged in two different communication sessions. The request session includes the invocation by the client of an operation passing the input parameters (*par1*, … *parN*, *jobGUID*, *notifyAt*) and the acknowledgement by the service that the request was received. A service shall return an acknowledgement when it is ready to respond to any further actions for that request from the orchestration system. See Figure 10.

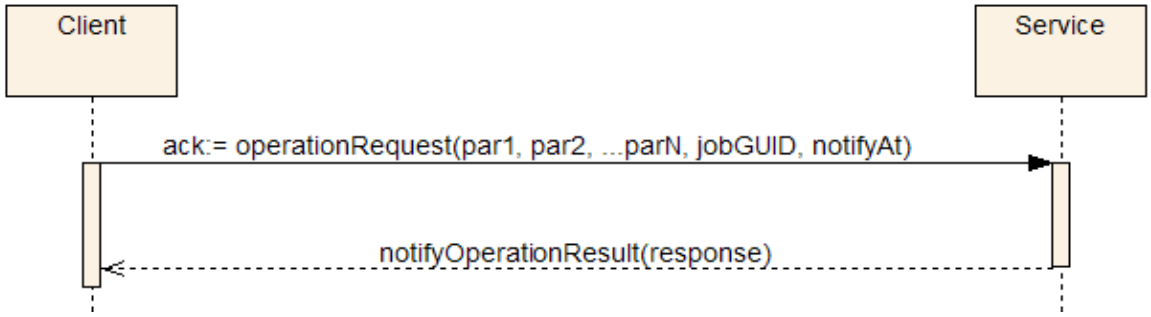

Figure 10: Asynchronous request/response with notification

The jobGUID parameter identifies uniquely the job request from the business process (the service client) point of view. The notifyAt parameter specifies to the service provider where to send the response message when the operation execution completes. It also specifies where to send an error notification message if the service fails during its execution. The notifyAt parameter shall be provided for the service to operate in this mode (see AsyncEndpointType definition).

A separate communication session is used to send the response message to the address specified by the replyTo element of the notifyAt parameter. If an error occurs during the process of the operation an error notification should be issued to the endpoint specified by the faultTo element of the notifyAt parameter.

An example of operation that employs this interaction mode is the transform operation of the Transform Media service. See Figure 11.

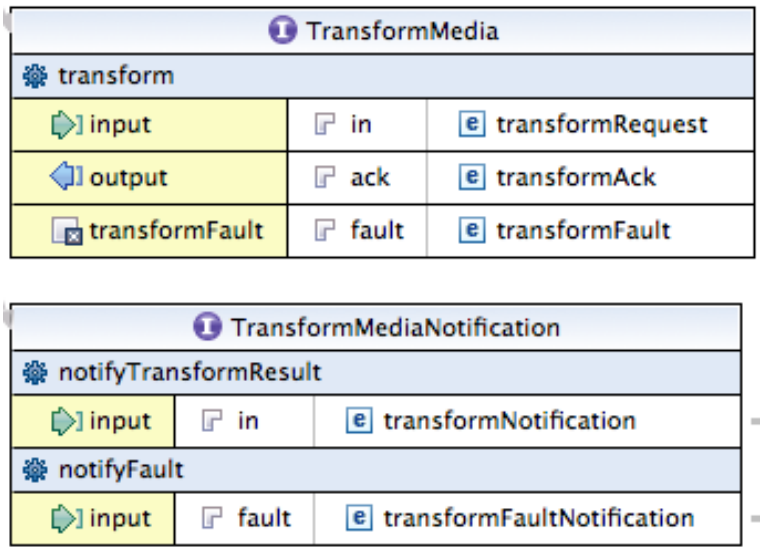

Figure 11: Example of asynchronous request/response with notification: Transform service

WSDL/SOAP requests shall have WSDL/SOAP notifications. RESTful requests shall have RESTful notifications.

For a RESTful asynchronous request, notifications shall be sent to the *replyTo* and *faultTo* endpoints provided by the defined by the RESTful notifications table provided with each service.

The body of a RESTful notification message shall be the service-specific notification type (e.g. CaptureNotificationType) for a reply and the service-specific fault type (e.g. CaptureFaultType) for a fault, as defined by the RESTful notifications table.

A RESTful notifications table shall have the following columns:

- **Description** – An informative description of the reason for the notification.
- **HTTP method** – The HTTP method that shall be used to send the notification (normally POST).
- **Body** – Type of the body of the notification that shall be defined by reference to the service-specific XML schema or base XML schema.
- **Generation events** – The events that cause a notification to be sent, such as job creation or job failure.

8.1.2.3 Asynchronous Request/Response with Polling – WSDL/SOAP

This is another interaction pattern for long running processes, where the preferred interaction pattern is not possible (e.g. a firewall preventing a service to call back a client). It is similar to the *Asynchronous Request/Response with Notification* interaction mode with the exception that a notification is not sent by the service when the service completes the operation. The *notifyAt* parameter shall not be provided for the service to operate in this mode. See Figure 12.

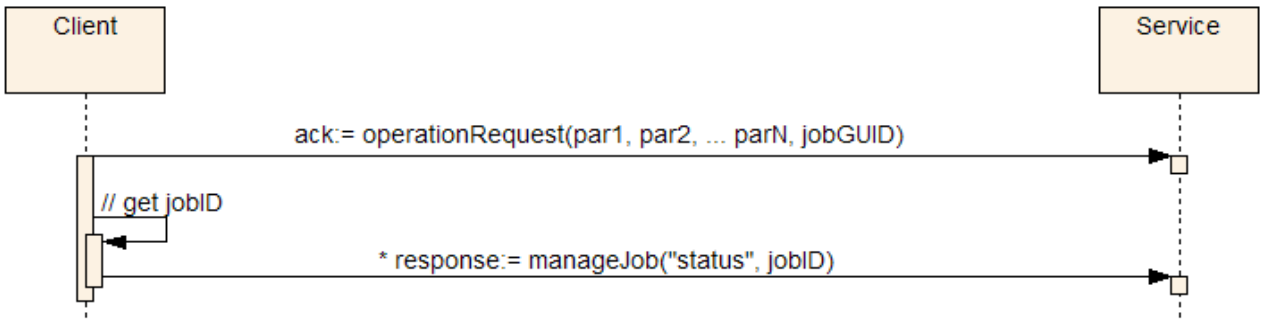

Figure 12: Asynchronous request/response with polling

When the client issues the request, it receives an acknowledgement message that contains the ID of the job as identified by the service.

In this mode of interaction, it is the responsibility of the client to poll the service, using the *queryJob* operation, to retrieve its queue status, running status and the result of the operation once it is completed. This is achieved using the *resourceID* of the job (that is part of the acknowledgment message). Once the job completes its execution the response message for the status request brings back the result of the operation.

The result of the job execution is contained in the *jobs* element. This element shall be present when retrieving the job information after the service completes the operation. The same information that would be part of the notification message in the pattern specified in Section 8.1.2.2 shall be present in this field.

A transform operation of the Transform Media service may be implemented using this type of interaction.

WSDL/SOAP implementation of FIMS services shall support either Notification or Polling.

8.1.2.4 Asynchronous Request/Response with Polling - RESTful

Every resource created by a service has a unique identity through which it can be addressed with a URI. This URI will have been returned in the *Location* header of the response message synchronously following the successfully created the resource. At any time between when a resource is created and deleted such that it has an addressable endpoint, the status of that resource shall be made available through a HTTP GET request to the URI for that resource.

8.1.2.5 Level of detail – all RESTful request/response types

The amount of detail returned by the GET request for a resource shall be configurable using a *detail* query parameter as part of the URI. This performs a similar function to the *BMContentFilterType* used in some SOAP calls. This parameter shall support at least the following three levels for all resources:

- *link* or *min*- Properties of the *ResourceReferenceType* only, providing just enough information to locate and identify the revision identifier for a resource.
- *summary* - Important properties of the resource itself and those of any types that the resource extends, as defined by a service implementation. All collections of elements shall be collapsed so that the items in the collection are displayed as links only.
- *full* - Full details of the resource including all properties with data, however important, and with all collections fully expanded so that every item is also provided with full detail.

The following three levels should be supported for *BMContentType* resources:

- *metadata* - The operation will return the matching contents with the metadata fields only populated.
- *physical* - The operation will return the matching contents with the physical (essence) fields only populated.
- *specific* - The operation will return the matching contents with the specific pre-defined fields only populated. The specific filter is defined as part of the RCR (*Repository Capability Registry*).

For example, here is the URI for a GET request to get the details of a capture job, with only the most basic of details returned:

```
http://capture.device7.acme.com/fims/job/AF7E4D2F-E981-4F66-B157-2026821E3102?detail=link
```

If the job is known to the service, a *200 OK* response is generated as follows:

```
<cms:captureJob ...>
  <bms:resourceID>urn:uuid:AF7E4D2F-E981-4F66-B157-2026821E3102</bms:resourceID>
  <bms:revisionID>2</bms:revisionID>
</cms:captureJob>
```

If the job is not known to the service, a *404 Not Found* response is generated, containing a fault message:

```
<cms:captureFault ...>
  <bms:code>DAT_S00_0003</bms:code>
  <bms:description>Invalid jobID - the supplied jobID does not exists.</bms:description>
  <bms:detail>The given job identifier 'AF7E4D2F-E981-4F66-B157-2026821E3102' is not
    known to this device.</bms:detail>
</cms:captureFault>
```

All the jobs currently known to a service may be listed using the job resource with the job identifier omitted.

Query parameters skip and limit may be used to control the pagination of collections of results, such as a *listjobs*:

- *skip* - Skip over the given number of record before starting a page of results. The default value when the parameter is omitted is zero, in other words the default behaviour is to skip no results.
- *limit* - Limit the number of results to the given number per response. The default value for the limit is service specific.

A service shall support one or both of XML and/or JSON representations. Where both formats are supported, a client may choose a format using an *Accept* header with the MIME type of preferred response format. A service that implements the requested representation shall respond with a message of that type, otherwise it shall respond with a fault.

8.1.3 Input and Output Media or Metadata

Media services often deal with media files. These services may consume and/or produce files that represent media essences. References to these media files are passed in the input and output messages for these services. Media is represented by a Business Media Object (BMObjectType), as described in the schemas (see "baseMediaService.xsd").

8.1.3.1 Processing the Input Media and Metadata

There are operations that require only a media essence (or list of essence), like transforming media content. In this case the service receives a container object (of type BMObjectType) with a list of content objects that extends the abstract type BMContentType, which represents the media essence(s) the service should operate on.

The definition of some operations may be complemented with metadata provided in the request payload.

8.1.3.2 Producing the output Media and Metadata

Services such as Transform Media produce new media essence (or a list of media essence). These services return one or more content objects (of type BMContentType) as a result of the operation.

Other services like Quality Assessment (QA) and Automatic Metadata Extraction (AME) return exclusively metadata either in the response payload or as a reference to a file containing the data.

8.1.4 Error and exception handling

Errors and exceptions detected by the service during the execution of job requests shall be returned to the service requester. For synchronous operations as described in Section 8.1.2.1 a fault message shall be returned as a response to the request. See Figure 13.

Figure 13: Fault messages handling during job execution

The fault message may contain detailed information about the error. See Figure 14.

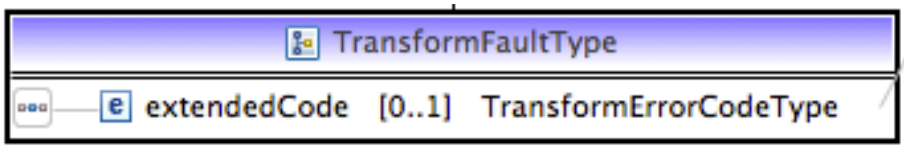

Figure 14: Example of fault message handling

The FIMS framework defines the enumeration of error codes and their description for each one of the service categories. A number of errors are common to all service categories and are presented in the FaultCodeType of the base schema.

For asynchronous operation, as defined in Sections 8.1.2.2, 8.1.2.3 and 8.1.2.4, errors may occur in two distinct phases. An error or exception may be thrown at the request time (e.g. Invalid Request Parameters) and a fault message shall be returned immediately as a response to the request. The behaviour is similar to the synchronous scenario.

On the other hand if an error is detected during the execution of a long-running process (e.g. job failed) and the *notifyAt* parameter has been set in the request message, then an error notification message shall be sent to the destination specified by *faultTo* property of *notifyAt*. See Figure 15.

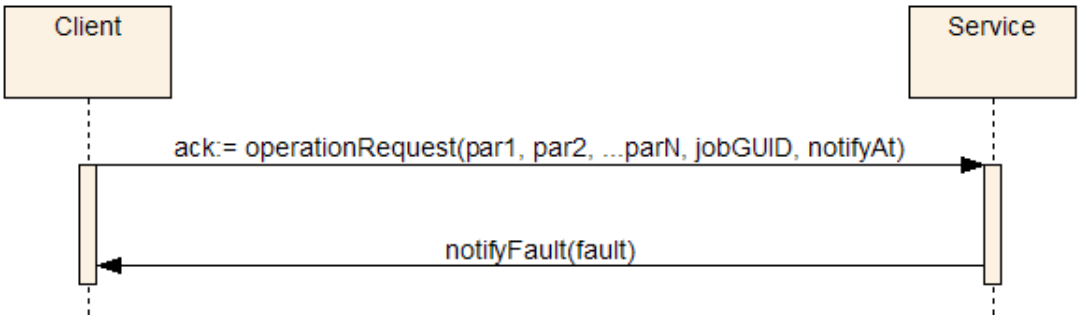

Figure 15: Fault message handling during execution of a long-running process

The recipient of the error notification message shall implement an interface defined by the service. For example for WSDL/SOAP, in the case of the *Transform Media* service the recipient implements the *notifyFault* operation specified by *TransformMediaNotification*. See Figure 16.

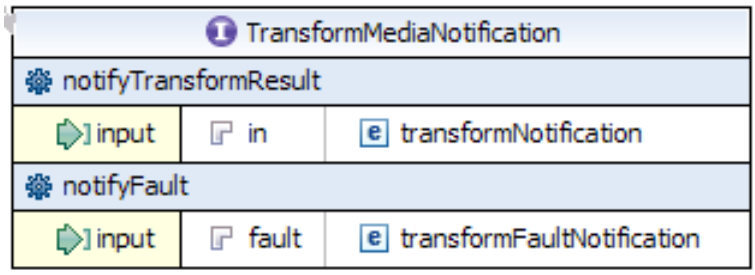

Figure 16: Example of fault message handling with notification

8.1.5 Failure Recovery

A service failure during the execution of a workflow should be a recoverable process. The framework allows for the retry of the failed service since in many cases the cause of the failure can be easily fixed by human intervention. In this scenario the workflow is paused at the invocation of the failed service and can be restarted manually after the service is fixed.

The framework also allows for the definition of a compensation mechanism in the workflow to rollback to a well-defined condition when the failure cannot be recovered by a simple retry mechanism.

8.1.6 Job Queue

A service may implement a queue to support multiple simultaneous requests. If implemented, job requests are en-queued in the order of priority and arrival. The service de-queues the jobs and processes them one by one. Multiple jobs can be de-queued at once if the service supports the execution of multiple simultaneous jobs (multi-threading).

Figure 17 shows the states associated to the job queue and the transitions initiated by the queue commands issued to the service. The states are associated to the processes that en-queue (accepts new jobs) and de-queue (starts execution of an en-queued job) requests. The *Started* state means that both the en-queuing and de-queuing processes are active. *Locked* means jobs cannot be en-queued but they are still being de-queued. *Stopped* means jobs are not being either en-queued or de-queued.

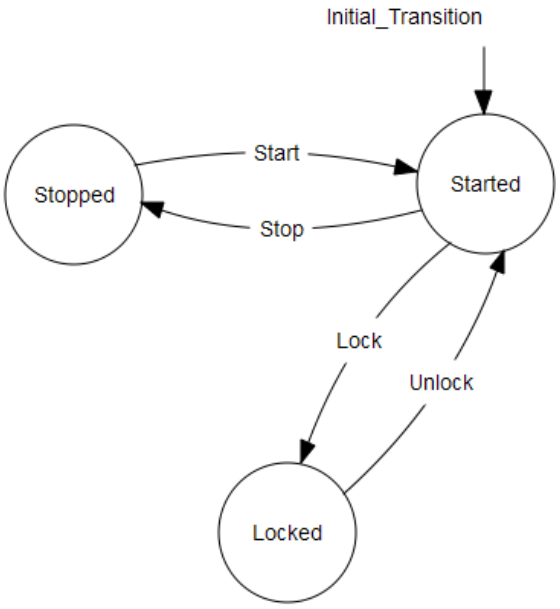

Figure 17: States associated to the job queue and the transitions initiated by the queue commands

The service shall implement the *manageQueue* operation to process queue commands. In case the service does not implement a queue it shall respond with “Feature Not Supported” error code (SVC_S00_0015).

A job queue may have a maximum size and when it is reached no more jobs will be accepted. A fault indicating the queue is full shall be thrown by the service when a client issues a new job request (SVC_S00_0008).

8.1.7 Job Execution Priority

The service should execute requests in the order of priority. The priority of a job is indicated in the

request message. Higher priority jobs take precedence over lower priority jobs.

The framework defines levels of priority for the request. The list of possible values for priority is: "low", "medium", "high", "urgent" and "immediate".

- A new job with "low" priority shall be allocated at the end of the queue.
- A new job with "medium" priority shall be allocated for execution prior to any "low" priority job but after existing "medium" priority jobs.
- A new job with "high" priority shall be allocated for execution prior to any "medium" and "low" priority job but after existing "high" priority jobs.
- A new job with "urgent" priority shall be allocated for execution prior to any "high", "medium" and "low" priority job but after existing "urgent" priority jobs.
- A new job with "immediate" priority should be executed as soon as the job request is received.

If the service can process only one job at a time and the job being executed has a lower priority than a new requested job, the existing job shall continue its execution till completion and only then the new job processing should start.

Except for a job with “immediate” priority, prioritization shall only affect jobs that are in the queue. To change what is actually running shall require an explicit operation, which is out of the scope of this specification.

8.1.8 Media Referencing

FIMS media services operate on content essence. In order to describe interoperability of content essence, a unified mechanism to reference essence and metadata is required. This Section describes an object model that provides a simple mechanism to support the FIMS requirements: the Business Media Object (BMObject). A BMObject provides business-level information on the media content exchanged by FIMS services (such as location of the media, format and size). The BMObject provides visibility at the business process level of the media content, thus the term 'business' - to indicate that this is a business-level object, which does not overlap with existing standards for the description of media and metadata, but rather provides a convenient mechanism to interoperate transparently with such standards. The BMObject can be as simple as a URL to the media content and a description, yet it is very flexible, since it can represent complex content models and provides a mechanism for transporting metadata as needed.

The BMObject model provides a simple reference to content essence files based on the BMContentType, and a minimum set of descriptive and technical metadata properties, based on EBU Tech 3293 (EBUCore Metadata Set). This can be used by the FIMS framework to make business rules driven decisions, such as selecting the best transform media service for a job based on the content format or the size of the content.

Future extensions of the framework may extend the BMObjectType with different kinds of collections of content. For example:

- Different editorial cuts of the same television programme.
- A complex mapping of the relationship between input and output content.

8.1.8.1 Partial Media Reference

From FIMS v1.2 onwards, three means of referencing partial media are provided, for example between a timecode start and end:

1. The Capture service has parameters for in points and out points. This allows the capture

device to be controlled so that it only captures part of a source.

2. Services including Transfer and Transform have contentPartAtom, simpleEDLAtom and wholeContentAtom elements that can be used to specify input media for a job that may be only part of the source material.

3. BMContentType has a nested timelines property of TimelinesType, where each segment (SegmentType such as BMContentPartType) of the timeline can be used to describe metadata about a time-bounded part of the content, including its original source and/or descriptive metadata.

8.1.9 Jobs

An essential aspect for the management of long-running media operations is the ability to check the status and interact with requested jobs. For example, the following tasks might be performed for transcoding operations:

- Check the status of a job or list of jobs
- Cancel an active job
- Pause an active job
- Stop an active job
- Change priority of a job
- Clear a queue
- Lock/unlock a queue

The adoption of a common interface for the status operations enables the use of common front-end tools for media services management, and the ability to interact with running tasks from a client or workflow. Queue and Job management types provide a mechanism to query and manage the status of jobs and job queues.

8.1.10 Errors

The base schema defines a fault type that can be extended by specific FIMS service classes to provide service-class specific error codes. One of the benefits in utilizing a media service abstract class is the definition of a common set of errors. This definition allows the client requesters to implement different error handling logic for each individual service provider used. With this approach, the client can implement a general compensation or error handling logic for all service providers that support a FIMS service specification.

8.2 Service Interface Overview

This section gives two common functions of service interfaces: Time Constraints and Profiles.

8.2.1 Time Constraints

8.2.1.1 Concept

Time Constraints are time-related constraints during the job execution that are specified in the service request message. Time Constraints consist of the following four parameters:

1) startJob

startJob is one of the elements of the JobType, and is applicable to any service. It specifies the

system time when the job shall start. Usually, this indicates the time at which a request message is extracted from the queue and is moved to the running state.

There are the following three types to specify the time parameter of startJob:

- NoWait type: execute immediately
- Time type: the time to start
- Latest type: the latest time at which a process can be started at the startProcess properly.

Supporting types of Time and/or Latest implies that the service supports a special queue which has ability to schedule the job start time according to the value of startJob. A service which supports Time or Latest type shall declare this in the Service Description.

2) startProcess

startProcess is a parameter for services that need to handle a real-time stream function such as Capture or Playout, and specifies the system time at the start of the stream process.

There are the following four types to specify the time parameter of startProcess, which may need to be frame accurate:

- NoWait type: execute immediately
- Time type: the time to start
- TimeMark type: the time at which the TimeMark embedded in essence such as a timecode is detected
- ServiceDefinedTime type: the time defined by a service

A service which supports Time and/or TimeMark type shall declare this in the Service Description.

3) stopProcess

stopProcess is a parameter for services that need to handle a real-time stream function such as Capture or Playout. This specifies the time when the stream process shall stop.

There are the following five types to specify the time parameter of stopProcess, which may need to frame accurate:

- OpenEnd type: the time at which a stop command is received
- Time type: the time to stop
- TimeMark type: the time at which the TimeMark embedded in essence such as a timecode is detected
- Duration type: the time at which point the specified duration has elapsed since startProcess
- ServiceDefinedTime type: the time defined by a service

A service which supports Time, TimeMark, and/or Duration type shall declare this in the Service Description.

4) finishBefore

finishBefore is one of the elements of the JobType, and is applicable to any service. It specifies the time by which the job shall have been completed.

finishBefore also specifies the deadline for the job execution. For example, in the case of specifying TimeMark type in the stopProcess, it can be used as a timeout time. When finishBefore is

exceeded before the job is completed, the service shall notify "SVC_S00_0016: Deadline passed" to the orchestration system.

If the service cannot accommodate the time constraint, the service shall notify the error code SVC_S00_0017 to the orchestration system: “Time Constraints in request cannot be met”.

A service that can control the job completion time by using finishBefore shall declare this in the Service Description. Even if a service does not itself support the function to control the job completion time, an orchestration system should specify finishBefore as a deadline.

8.2.1.2 Use Cases on Time Constraints

Table 1 shows possible use cases with the combination of "startProcess" and "stopProcess".

Table 1: Time constraints using a combination of "startProcess" and "stopProcess"

startProcess	stopProcess	Description
NoWait	OpenEnd	Start ASAP, stop when the stop command (manageJobRequest) is received.
	TimeMark	Start ASAP, stop when the specified timeMark is detected.
	Time	Start ASAP, stop at the specified time.
	ServiceDefined Time	Start ASAP, stop at the service defined time.
	Duration	Start ASAP, stop when the specified duration has passed.
TimeMark	OpenEnd	Start when the specified timeMark is detected, stop when the stop command (manageJobRequest) is received.
	TimeMark	Start when the specified timeMark is detected, stop when the specified timeMark is detected.
	Time	Start when the specified timeMark is detected, stop at the specified time.
	ServiceDefined Time	Start when the specified timeMark is detected, stop at the service defined time.
	Duration	Start when the specified timeMark is detected, stop when the specified duration has passed.
Time	OpenEnd	Start from the specified time, stop when the stop command (manageJobRequest) is received.
	TimeMark	Start from the specified time, stop when the specified timeMark is detected.
	Time	Start from the specified time, stop at the specified time.
	ServiceDefined Time	Start from the specified time, stop at the service defined time
	Duration	Start from the specified time, stop when the specified duration has passed.
ServiceDefined Time	OpenEnd	Start at the service defined time, stop when the stop command (manageJobRequest) is received.
	TimeMark	Start at the service defined time, stop when the specified timeMark is detected.
	Time	Start at the service defined time, stop at the specified time.
	ServiceDefined Time	Start at the service defined time, stop at the service defined time.
	Duration	Start at the service defined time, stop when the specified duration has passed.

8.2.1.3 Sequence Diagram Examples

Figure 18 shows an example of the Capture Service sequence diagram in terms of Time Constraints. The source type of this example is VTR and both inPoint and outPoint are the specified time code.

A service receives a request message from an Orchestration System. The service starts job at startJob, and performs some actions such as cueing up to the inPoint, starting playback and detecting specified time code. An important point to note is that the instruction of how and when to control the device is not included in the request message. The service manages to control the

device by itself according to the information such as startProcess/stopProcess, inPoint/outPoint in the request message. A capture process starts at the serviceDefinedTime; it will stop at another serviceDefinedTime. After some finishing job process is performed, a CaptureNortification is issued to the Orchestration System to report completion.

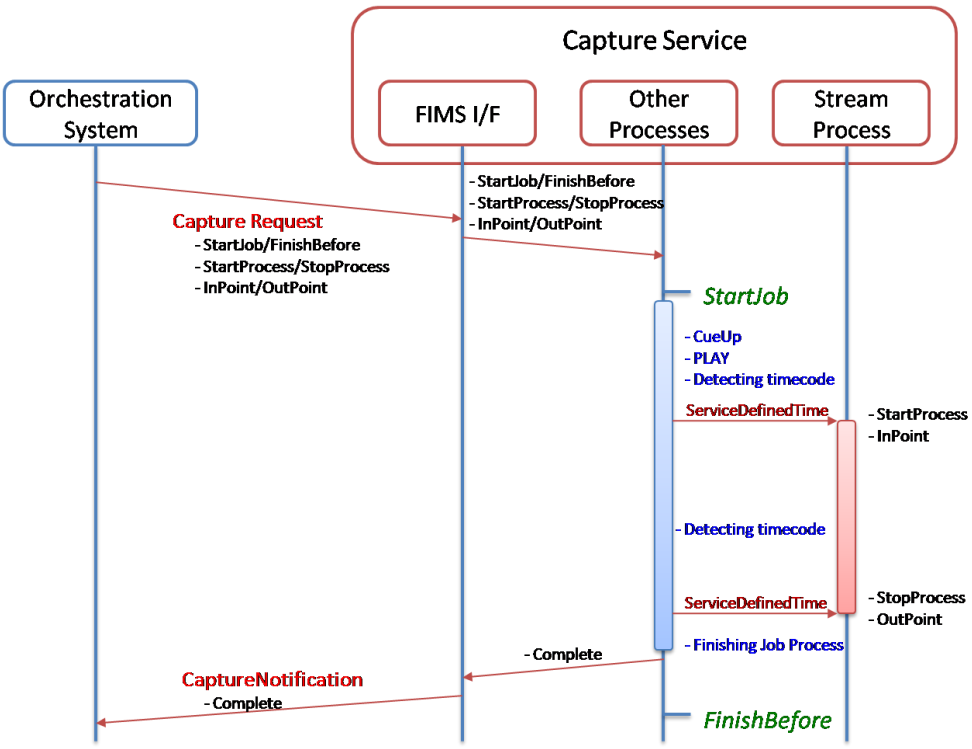

Figure 18: Example of the Capture service sequence diagram in terms of time constraints

Figure 19 shows the case where Time Constraints in the request message cannot be met for some reason. This can occur when a service receives the request message, or during queuing, or even during running. In this case, the service shall issue a CaptureFault with the error code “SVC_S00_0017: Time Constraints in request cannot be met" to the Orchestration System.

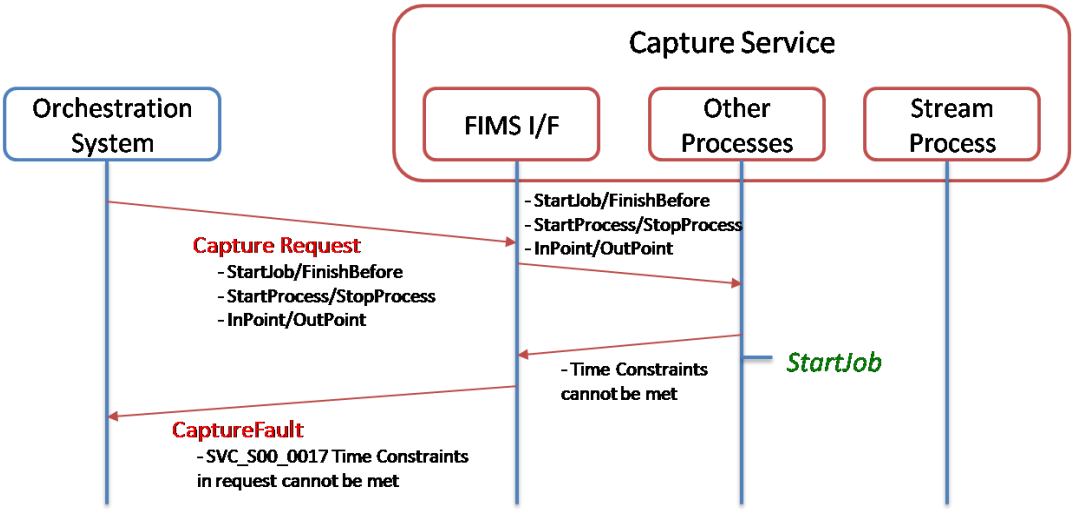

Figure 19: Example where Time Constraints cannot be met

Figure 20 shows the case where the time specified by finishBefore has passed before the capture process has been completed. In this case, the service shall issue CaptureFaultNotification with the error code "SVC_S00_0016 Deadline passed" to the orchestration system. After issuing the CaptureFaultNotification, the service should wait for the next instruction from the orchestration without stopping the process.

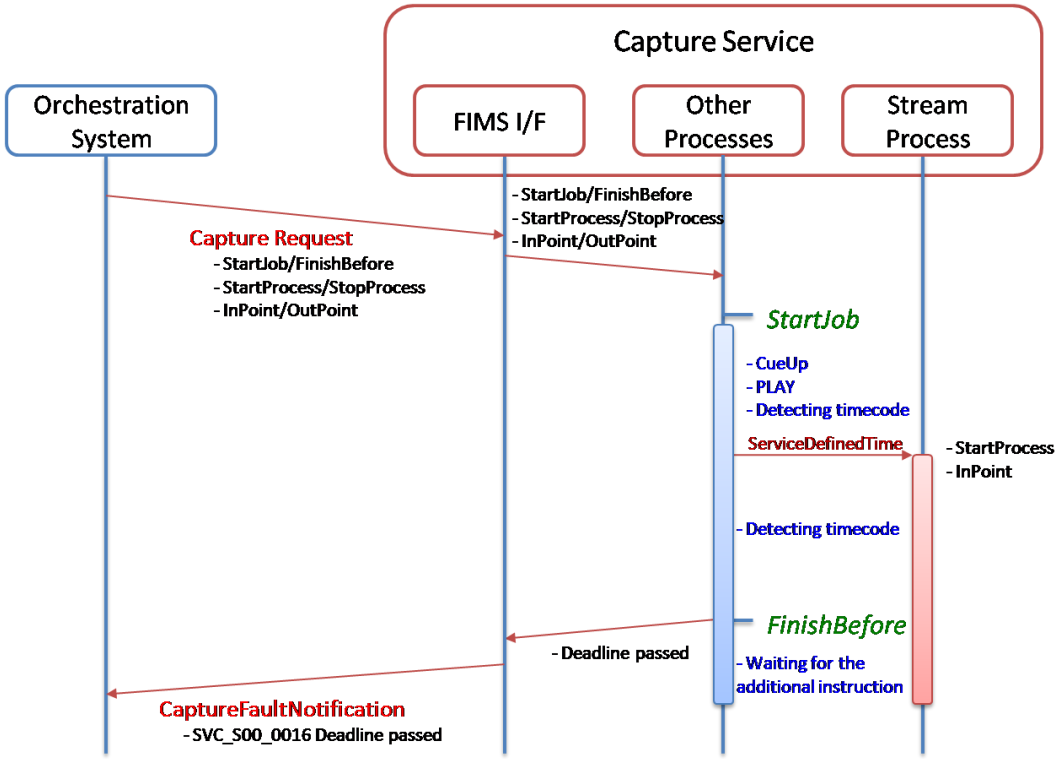

Figure 20: Example where the time specified by finishBefore has passed before completion

8.2.2 Profiles

Requests to any FIMS service interface shall include a profile that describes the specific operations that will take place as part of that request. Profiles themselves are composed of parameters specific to that service interface and generic elements that are available for re-use between interfaces. These reusable elements are combined into groups referred to as Atoms.

Profile structures are specified for each service interface that FIMS defines. Profiles inherit from *ProfileType*, which is used to provide a common basis for all profiles (Figure 21).

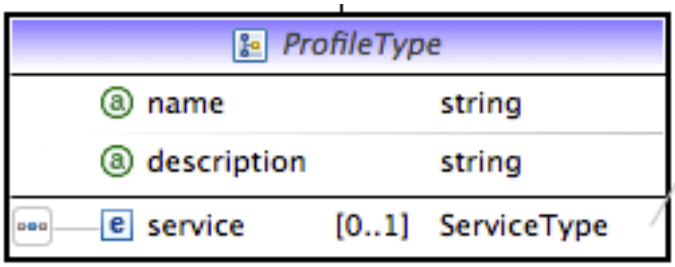

Figure 21: ProfileType

Profiles can be shared between jobs and included by reference. See Section 6.2.3.

Where possible, implementations should use the specified profile.

Figure 22 shows an example of a request with two Capture profiles being used to create main stream essence (J2K + MXF) and proxy essence (AVC + MP4) and outputting though different transfer atoms (possibly to different locations over different transports).

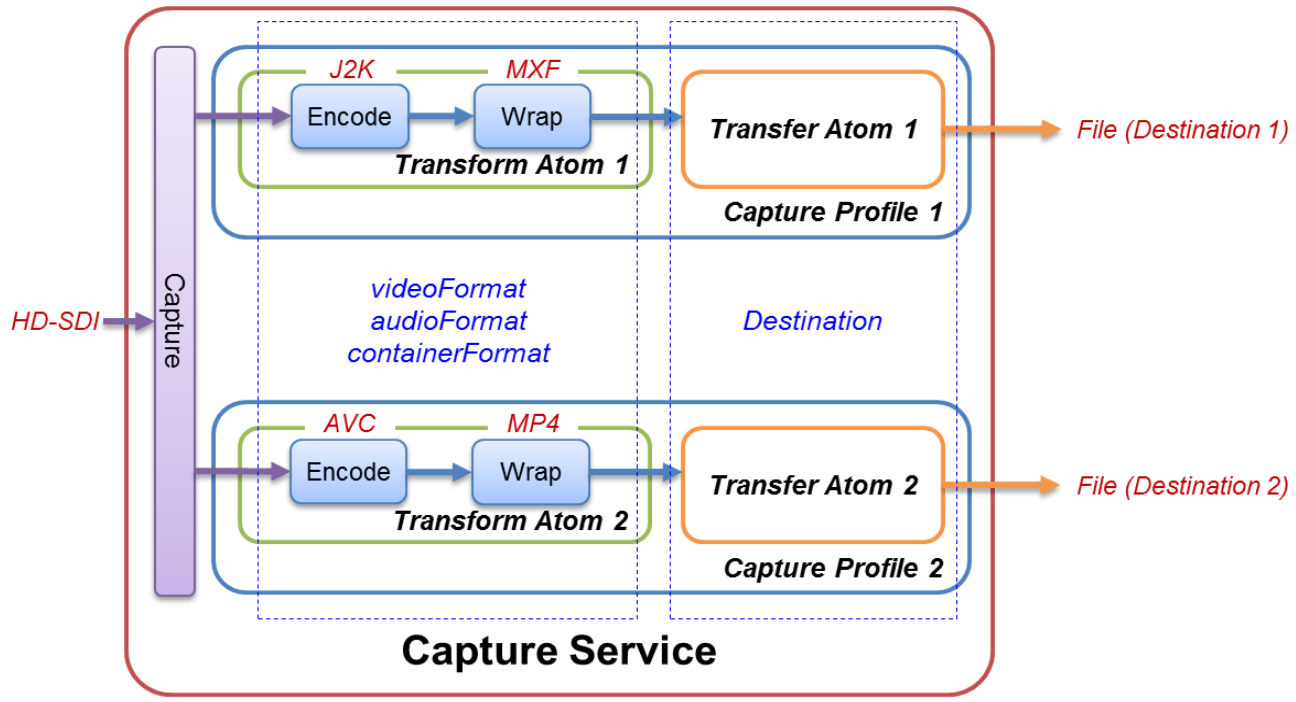

Figure 22: Example of Capture request with multiple profiles

8.3 RESTful service interfaces

This Section defines the RESTful representation of FIMS values in REST messages and how RESTful services are defined. The rules of this Section are to a REST services what the WSDL files are WSDL/SOAP services.

REST service definition tables shall be used to define FIMS RESTful services. These tables shall be included as documentation with the XML schema documentation for each service. Each table makes reference to elements of the schemas that define the XML or JSON representation of the body of a REST message.

8.3.1 Namespaces for REST service definition

Table 2 defines the namespaces and prefixes that shall be used for the definition of REST services.

Table 2: The namespaces used for the REST service definition in the XML schema

Schema name	Prefix	Namespace
Base media service	bms	http://base.fims.tv
Base time values	tim	http://baseTime.fims.tv
Transfer media service	tms	http://transfermedia.fims.tv
Transform media service	tfms	http://transformmedia.fims.tv
Capture media service	cms	http://capturemedia.fims.tv
Repository service	rps	http://repository.fims.tv
Quality analysis service	mqas	http://mediaqa.fims.tv

The namespaces listed in the table are for the services defined as of this version of the specification. Any services that make use of this framework, but that are not part of this FIMS version, shall state the namespace and prefix used to define the RESTful mapping of the service.

8.3.2 REST service definition tables

The documentation for each service shall include a REST *service definition table* that defines how the operations of the service are represented as REST messages. The table shall consist of the following columns:

- **Description** – Description of the operation.
- **HTTP method** – HTTP method that shall be used for a request message for the operation, e.g. GET, POST, DELETE.
- **URI** – URI that shall be used to access the RESTful service that is specified relative to the endpoint for the REST service, e.g. "…/job".
- **Request body** – Reference to the XML element name and XML element type that shall be used to represent the FIMS value that shall be the body of the request message.
- **Success body** - Reference to the XML element name or XML element type that shall be used to represent the FIMS value returned as the body of a response message for a successful operation.
- **Failure body** - Reference to the XML element name or XML element type that shall be used to represent the FIMS value returned as the body of a response message for a failed operation.
- **HTTP request headers** – In addition to the required version number header (Section 6.1.4.1) and standard HTTP headers, any FIMS-specific HTTP headers that shall be included with the request message.
- **HTTP response headers** – In addition to the required version number header (Section 6.1.4.1) and standard HTTP headers, any FIMS-specific headers that shall be included with a successful operation response message.

The status code of a response message shall either be a redirect code (300 range) or as follows:

- 200 – OK – Successful operation, synchronous or asynchronous response;
- 201 – Created – Successful operation that has resulted in the creation of a new object;
- A failed operation containing a fault message with the status code set to the value of status code as defined for the specific fault. See the general and specific FaultType documentation as part of the XML schemas.

FIMS REST clients should follow HTTP redirection responses as defined in the HTTP specifications.

Any URI parameters, HTTP headers, and query arguments shall be described as in Table 3. Parameters on a URI are identified in REST service definition tables using curly braces {} and require substitution with appropriate parameters before use. Query parameters will be *italicized*.

Table 3: FIMS REST tables template parameters

Parameter	Description	Values
{jobID}	Unique identifier for a job.	bms:ResourceIDType
{urlToJob}	On successful job creation, this value is the full path to the job	http(s)://{server name}/job/{jobId}
{revisionID}	Specific revision of content that is referenced in a repository. No revision specifies reference to the latest version. RevisionIDs are of NMTOKEN type.	http(s)://{server name}/content/abc…123/revision3
{repository credentials}	Authorization credentials for the repository.	HTTP header = X-FIMS-UserName HTTP header = X-FIMS-Password HTTP header = X-FIMS-SessionToken
{lock token}	Lock obtained from the repository.	HTTP header = X-FIMS-LockTokenId
{super token}	Master lock obtained from the repository.	HTTP header = X-FIMS-SuperLockTokenId
{JobFilterCriteria}	See Table 4 for job filter parameters.	

Query parameters that can be used to select jobs on query job services are detailed in Table 4.

Table 4: Job filter criteria

Parameter	Description	Type
jobInfoSelectionType	Return only mandatory attributes or all attributes of a job	"all" or "mandatory"
toDate	Jobs to be listed shall have started on or before the date specified in this field.	dateTime
fromDate	Jobs to be listed shall have started on or after the date specified in this field.	dateTime
includeQueued	A flag to indicate job or jobs in the queue.	Boolean "true" or "false"
includeFinished	A flag to indicate job or jobs in the 'Completed', 'Stopped' or 'Cleaned' state.	Boolean "true" or "false"
includeActive	A flag to indicate job or jobs in the 'Running', 'Paused' or 'Unknown' state.	Boolean "true" or "false"
includeFailed	A flag to indicate job or jobs in the 'Failed' state.	Boolean "true" or "false"
maxNumberResults	Maximum number of results to be listed.	Positive integer value

8.3.3 REST message XML representation

The REST service definition tables and RESTful notification tables specify the type of FIMS values to be carried in the body of messages by reference to XML schema elements and types. FIMS XML schemas define both the data type of FIMS values and their serialization as XML. The body of a REST message with "Content-Type" set to "application/xml" shall be values serialized to and deserialized from XML as defined by the referenced element of the FIMS XML schemas.

8.3.4 REST message JSON representation

8.3.4.1 JSON representation as conversion to and from XML

The body of any REST messages with "Content-Type" set to "application/json" shall be FIMS values serialized to JSON as defined by first serializing the value to XML as per Section 8.3.3 and then applying the rules of this section. In reverse, the rules of this section shall be applied to convert the JSON value back to XML and then deserialized according to the rule of Section 8.3.3.

Note: The conversion mechanism for JSON to and from XML is to provide a means to define the JSON representation using the existing XML schema. Implementations do not have to implement this approach and may use more efficient means of serialization and deserialization.

In this section, a name/value pair for an object in a JSON document is referred to as a JSON field.

Essential to the interoperability of FIMS services, the FIMS XML schema use XML namespaces to distinguish between fields that are: defined by the base schemas; specific to current services; specific to future services; extension properties. As JSON does not have an explicit concept of namespaces, FIMS JSON documents shall follow XML namespace rules in field names and shall carry all "xmlns" attributes as would be expected in a corresponding XML document. JSON field names should use prefixes wherever possible so as to unambiguously communicate the explicit namespace of the property in use.

Note: XML namespace rules will be followed as a consequence of applying the rules in this section. Frameworks that support serialization to and from both JSON and can use different approaches when dealing with namespaces. Care must be taken - and extra code may have to be written - to ensure full interoperability between JSON services. For example, the prefix of a field name may change depending on the "xmlns" namespace declaration currently in scope.

8.3.4.2 Documents

Exactly one FIMS XML document shall be represented by exactly one JSON document.

The root object of the JSON document shall contain exactly one field. This field shall represent the value of the root element of an equivalent XML document through the application of the *element rule*.

8.3.4.3 Element rule

Elements of simple type in XML constrained with XSD attribute *maxOccurs* set to 0 or 1 shall map one-to-one to JSON fields. The name of the field shall be the prefixed namespace-qualified name or context-based namespace-unqualified name as per its contextual use in a corresponding XML document. The value of the field shall be determined by application of the *simple type mapping rules*.

Elements of complex type constrained with XSD attribute *maxOccurs* set to 0 or 1 shall be represented in JSON as fields of their parent object of type object. By application of the *element* or *attribute rules*, each of the child nodes of the element, whether attributes or elements themselves, shall be mapped to child fields of the JSON object.

Elements of any type constrained with XSD attribute *maxOccurs* set to a value greater than 1, including those defined as unbounded, shall be represented as JSON arrays, where each value of the array shall be the result of applying the element rule. The order or sub-elements in an XML sequence shall correspond to the order of equivalent values in a JSON array.

Note: By definition, fields of a JSON object are out of order, whereas the sub-elements of a FIMS element have a specific order. It is possible that an application that translates a FIMS JSON document into a FIMS XML document will need to reorder the fields of the JSON objects according to the FIMS XML schema prior to validation.

For example:

1. XML element

```
<Fred>ginger</Fred>
```

is represented in JSON as

```
"Fred":"ginger"
```

.

2. XML element

```
<Fred>2</Fred>
```

is represented in JSON as

```
"Fred":2
```

.

3. XML element

```
<bms:Fred>ginger</bms:Fred>
```

is represented in JSON as

```
"bms:Fred":"ginger"
```

.

4. XML element sequence

```
<fromage>cheddar</fromage><fromage>Stilton</fromage>
```

is represented in JSON as

```
"fromage":["cheddar","Stilton"]
```

.

5. XML complex type element

```
<thing><child>stuff</child></thing>
```

is represented in JSON as

```
"thing":{"child":"stuff"}
```

.

6. XML complex type element

```
<thing about="that"/>
```

is represented in JSON as

```
"thing":{"@about":"that"}
```

.

8.3.4.4 Attribute rule

All XML attributes shall be JSON field values that are members of the parent element that defines them. The name of the field shall be the same as that used in for the XML with an '@' ("commercial at" symbol) prepended to the start. If the XML attribute name is namespace-qualified then the JSON field name shall be name namespace-qualified. If the XML name has no namespace qualification then JSON name shall have no namespace qualification. The type of the value shall be mapped according to the simple type mapping rules.

For example:

1. XML attribute

```
<... "fred"="ginger" ...>
```

is JSON field

```
..., "@fred":"ginger", ...
```

2. XML attribute

```
<... "bms:fred"="ginger" ...>
```

is JSON field

```
..., "@bms:fred":"ginger", ...
```

3. XML attribute

```
<... "fred"="3" ...>
```

is JSON field

```
..., "@fred":3, ...
```

8.3.4.5 Simple type mapping rules

Table 5 defines the mapping between simple types in the XSD schemas and the limited set of simple JSON data types for field values of number, string and Boolean.

Table 5: Simple type mappings from XSD types to JSON types

XSD type	JSON type
Boolean	JSON Boolean "true" or "false"
int, long, double, float, short, byte and derivatives	
such as non-negative int.	JSON number
string, NCName	JSON string
string constrained by enumeration or regular expression	JSON string with the same restriction
date, dateTime, time, duration	JSON string of the same format.

Annex 1: Future Visions (Informative)

A1. Pipelined Media Services

A Pipelined Media Services is composed of more than one existing service in order to realize functions and/or performance that use of individual existing services by themselves cannot achieve.

As described in Section 8.2.2, a pipelined media service within a service can be realized using profiles. Figure A1 shows an example of extended profile where two Capture (Transform) profiles are used to create main stream essence (J2K + MXF) and proxy essence (AVC + MP4) with AV Process.

AV Process is added in the transformAtom so that processing A/V essence is enabled during capture.

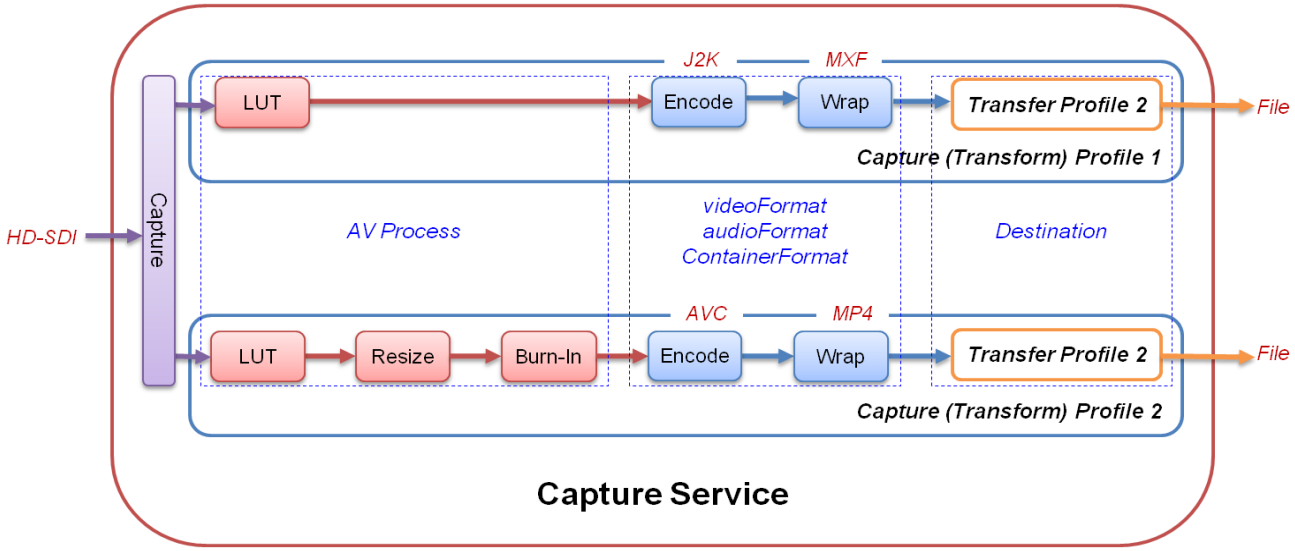

Figure A1: Example of pipelined media service