Date: January 2012

UML Profile for Advanced and Integrated Telecommunication Services (TelcoML)

FTF Beta 1

OMG Document Number: ptc/2012-01-02
Standard document URL: http://www.omg.org/spec/TelcoML/
Associated File(s)*:

	http://www.omg.org/spec/TelcoML/20110601
mars/11-06-07	http://www.omg.org/spec/TelcoML/20110601/TelcoML_CompositionProfile.xml
	http://www.omg.org/spec/TelcoML/20110602
mars/11/06-08	http://www.omg.org/spec/TelcoML/20110602/TelcoML_EnablerLibrary.xml
	http://www.omg.org/spec/TelcoML/20110603
mars/11-06-05	http://www.omg.org/spec/TelcoML/20110603/TelcoML_CompositionProfile.papyrus.uml http://www.omg.org/spec/TelcoML/20110603/TelcoML_EnablerLibrary.papyrus.uml http://www.omg.org/spec/TelcoML/20110603/TelcoML_EnablerLibrary.txt

---

* original files: Normative: mars/11-06-07, mars/11-06-08
Non-normative: mars/11-06-05

This OMG document replaces the submission document (mars/2011-08-06, Alpha). It is an OMG Adopted Beta Specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to *issues@omg.org* by May 21, 2012.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on September 21, 2012. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , IMM™ , MOF™ , OMG Interface Definition Language (IDL)™, and SysML™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. (as of January 16, 2006) at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

The objective of this specification is to define a UML Profile for designing advanced and integrated Telecommunication services.

An advanced and integrated telecom service generally means a service that exploits the convergence of communication networks - landline, wireless and voice, and in the same time takes advantage of the plethora of facilities accessible from the World Wide Web. Sensibility to user context - like presence, localization, user preferences and use of communication means (like SMS or voice messaging) are some of typical ingredients that appear in innovative telecommunication services that operators or third party service providers would like to offer to end-users. Application of model-driven techniques for an agile development of this new kind of services will be facilitated by the definition of a domain specific UML profile.

This specification standardizes firstly the UML representation of a selected set of service interfaces of typical telecommunication facilities (the TelcoML Enabler Library). Secondly, among different possibilities offered by UML to represent service logic, this specification selects one convention to represent service compositions (the TelcoML Composition profile).

The UML Profile defined by this specification is defined as a specialization of the SoaML UML Profile (see normative references) and is named Telecommunication Modeling Library (TelcoML).

2 Conformance

An implementation that claims conformance to this specification needs to refer to one or two of the two conformance levels defined here:

- TelcoML Editing: A UML tool that has pre-installed the model elements of the TelcoML profile: the interface definitions specified in the Enabler Library, the list of stereotypes of the Composition Profile. Such tool should provide means to edit state machines in line with TelcoML notation conventions defined by the Composition Profile. In addition such tool should provide export facility using either UML compliant XMI 2.1 or ALF representation.
- TelcoML Execution: A UML tool that supports TelcoML Edition compliance level and that provides in addition means to execute or simulate service compositions specified using the TelcoML composition profile. This implies means to connect service interfaces in the Enabler Library to concrete implementations of these telecommunication facilities.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

List of normative references

1. *Object Constraint Language 2.0*, OMG document number formal/2006-05-01
2. *Service oriented architecture Modeling Language (SoaML)*, OMG document number ptc/2009-12-09
3. *UML v2.1.2 Superstructure Specification*, OMG document number formal/07-11-02
4. Action Language for Foundational UML (ALF), OMG document number ptc/10-10-05.

5. *MOF v2.0 Specification*, OMG document number formal/06-01-01
6. *MOF 2.0/XMI Mapping v2.1.1 Specification* document number formal/07-12-01

Normative parts of TelcoML

The following is included as part of this specification:

- XMI Document for TelcoML mars/2011-02-03

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Unified Modeling Language (UML)

The Unified Modeling Language, an adopted OMG standard, is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecommunications, aerospace) and implementation platforms (e.g., JEE, .NET).

SoaML Specification – FTF Beta 2 10

Modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecommunications, aerospace) and implementation platforms (e.g., JEE, .NET).

XML Metadata Interchange (XMI)

XMI is a widely used interchange format for sharing objects using XML. Sharing objects in XML is a comprehensive solution that builds on sharing data with XML. XMI is applicable to a wide variety of objects: analysis (UML), software (Java, C++), components (EJB, IDL, CORBA Component Model), and databases (CWM).

EXtensible Markup Language (XML)

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. RDF and OWL build on XML as a basis for representing business semantics on the Web. Relevant W3C recommendations are cited in the RDF and OWL documents as well as those cited under Normative References, above.

One API Specification

GSMA OneAPI initiative defines a commonly supported set of lightweight and Web friendly APIs to allow mobile and other network operators to expose useful network information and capabilities to Web application developers. It aims to reduce the effort and time needed to create applications and content that is portable across mobile operators.

5 Symbols

UML: Unified Modeling Language

MOF: Meta Object Facility

SoaML: Service Oriented Modeling Language

SMS : Short Message Service

MMS: Multimedia Message Service

6 Additional Information

6.1 Acknowledgements

The following persons were mainly responsible for the specification:

Mariano Belaunde (Orange Labs), Irv Badr (IBM), Jenny Huang (AT&T), Huascar Espinoza (ESI), Sumeet Malhotra (Unisys), Hendrik Berndt (DOCOMO Euro Labs)

The following companies submitted this specification:

- International Business Machines
- Unisys

The following companies supported this specification:

- France Telecom - Orange Labs
- AT&T
- European Software Institute
- DoCoMo Communication Laboratory Europe GmbH
- SINTEF
- Telefonica

7 TelcoML Overview

7.1 Introduction

This clause is normative except sub clause 7.3. This clause gives an overview of TelcoML, a UML profile for advanced and integrated telecommunication services built on top of SoaML language.

SOA is an architectural paradigm for defining how people, organizations and systems provide and use services to achieve results. SoAML is a standard extension to UML 2 that facilitates these services modeling. The TelcoML provides additional extension to SoAML with the consideration of real-time communication services and many of the existing communication services and architecture standards.

The value proposition of the TelcoML is to provide a common abstraction to all existing communication services standards so that tools can be built for the Communication Service Providers (CSP) to model variety of services in a consistent manner.

The TelcoML specification consists of the following normative constituents:

- The TelcoML Enabler Library: A set of SoaML interfaces representing telecom specific facilities. It comprises management interfaces for services and various interfaces for communication facilities (like SMS messaging, presence and so on). The interfaces included here are generally re-formulations of existing APIs already standardized in different telecom-oriented standardization bodies (mainly TMF, GSMA, OMA).
- The TelcoML Service Composition Profile: a specialization of SoaML UML profiles to enable specifications of composite services with enhanced capacities like voice interaction support. UML State Machines are used here with some specific notational convention (transition centric notation, additional icons).

7.2 Relation with other standards

This specification has strong connection with the following standard coming from various standardization bodies:

- **OASIS** - OASIS SOA Reference Model provides the foundation for SoaML, so transitively it is also fundamental to TelcoML, especially for IT capabilities for Service Component definition and combination
- **TM Forum** - The Software Enabled Services (SES) Management Interface (SMI) - which is actually in draft state - will be considered for integration in future releases of the TelcoML library. Annex A provides a temporary non-normative SoaML representation of this API. Besides that TM Forum Frameworx defines an architecture for service management Along with SES Reference Architecture and other related work such as IPsphere for B2B interactions of service trading. The TM Forum spec. will help to complete the picture of service lifecycle management (LMM) from concept, design, deploy, operation to retirement.
- **OMA** - OMA is standardizing various telecom enablers (like Presence). Some of them have been reformulated as UML interfaces and integrated to TelcoML library. Notice however that TelcoML only retain functional aspects of enablers definitions (protocol aspects are skipped)
- **GSMA** - The One APIs from GSMA is the successor of Parlay/ParlayX APIs. One APIs are reformulated in terms of UML and integrated in TelecoML.
- **W3C** - The VoiceXML standard defines concepts and XML vocabulary for execution of voice based interaction. Some voice based concepts are present in TelcoML in the IVR Facility enabler (Clause 8.7) and in the composition profile in Clause 9.4.2. VoiceXML represents an implementation technology to support voice features exposed by TelcoML.

7.3 Service Delivery Environment

In this sub clause we recall architectural considerations of a service delivery environment, with service delivery platforms (SDPs) at its core. SDPs will typically provide a framework that consists out of capabilities for service creation, composition, execution and service control. SDPs will consist out of service enabler components that are either service independent, service specific or both. Examples of service delivery support functions that an SDP may offer are shown in Figure 7.1. The classification of supporting function follows TM Forum Solution Frameworks (Frameworx).

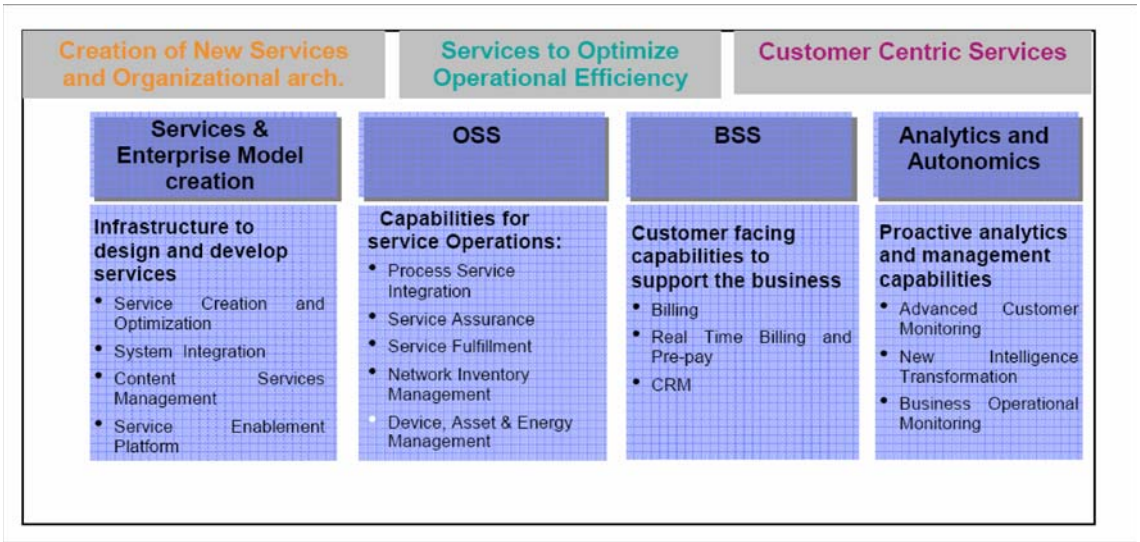


Figure 7.1 - Example service delivery supporting functions

Most of the service interfaces included in the TelcoML Enabler Library (Clause 8) fit in the first category: creation of new services. An exception is the SES SMI interface which fits in the BSS part. The Composition Service Profile (Clause 9) fits in the first category since its purpose is the specification of executable composite services.

Note: In future releases of this specification other kind of interfaces could be incorporated, like the OneAPI Billing interface as well as additional interfaces coming from OneAPI v2.0 (like user data management).

8 TelcoML Enabler Library

8.1 Introduction

This clause is normative, with the exception of some sub-sections explicitly marked as non normative.

The TelcoML Enabler library defines a set of predefined service interfaces to facilitate the definition of composite services accessing well-known telecommunication facilities (called enablers) usually provided by telecom operators. These service definitions are provided in a sufficient technology agnostic way to enable independence in respect to the enabler provider and to the usage of a specific implementation technology.

Whenever possible the service interfaces defined here are reformulations in UML of pre-existing enablers defined in telecom-oriented standardization bodies and implemented by telecom operators.

8.2 Conventions

Each enabler is described using at least two sub-clauses:

- The “Overview” gives a short explanation of the purpose of the enabler.
- An optional “Context” provides additional information that may be of interest to understand the design of the interface. If present, this is non normative.
- The “Interface Definitions” gives the list of interfaces defined for the enabler.

For each service interface, there is a specific section, containing:

- The “Data Types” lists the data types in graphical form and with additional textual explanations.
- The “Operations” provides the details on each service operation:
  - ·firstly a header providing in short the semantics of the operation,
  - ·secondly the complete signature (with all multiplicities),
  - ·then, if useful for the comprehension, an additional explanation of the purpose of the operation,
  - ·finally three sections: parameters, outputs and exceptions.

8.3 Package Structure

Technically the TelcoML Enabler Library is defined by a UML Package containing a list of sub-packages corresponding to each sub clause (“Generic Messaging,” “Short Messaging,” and so on). The SoaML profile applies to this package. The ServiceInterface stereotype used in the enabler library is the stereotype defined by SoaML.

8.3.1 Generic Messaging

8.3.1.1 Overview

This enabler provides a simplified interface to send messages using different medias (SMS, SMS, EMAIL, Instant Messaging and Voice). In contrast with complete SMS and MMS facilities (see next sub clauses), this enabler only allows using individually one of the messaging methods with a limited set of parameters.

8.3.2 Interface Definitions

8.3.2.1 Messaging Interface

The Messaging interface provides one operation for each communication mode.

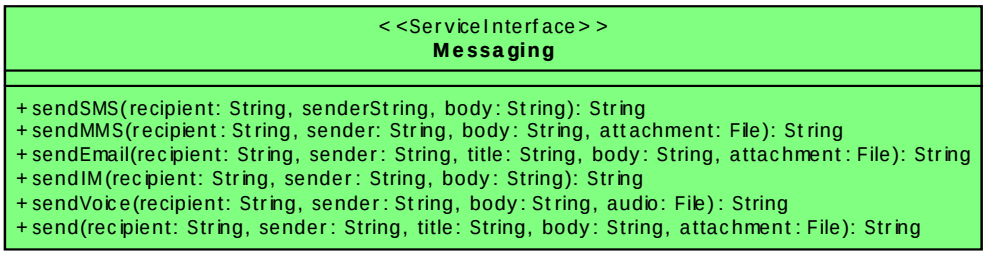


Figure 8.1 - Generic Messaging Service Interface

This interface does not require the definition of specific data types (only primitive types are used). File represents a file location provided as a string.

8.3.2.1.1 Operations

Send an SMS

`sendSMS(recipient: String, sender: String, body: String): String`

Sends an SMS to a recipient (example: a phone number). Success return implies that the message was successfully precessed but actual delivery may occur at a future time.

Parameters:

recipient : The phone address that will receive the SMS.
sender : The sender indication to be displayed by the receiving terminal.
body : The content of the short message.

Outputs:

A string status is returned: 'OK: <message-id>' or 'FAILED: <message reason>'

Exceptions:

InvalidRecipient : Raised if the recipient is not valid.

Send an MMS

`sendMMS(recipient:String,sender:String,body:String,attachment:File):String`

Sends an MMS to a recipient (example: a phone number).

Parameters:

recipient : The phone address that will receive the SMS.
sender : The sender indication to be displayed by the receiving terminal.
body : The content of the short message.

attachment : The file reference (a URL) to include in the message.

Outputs:

A string status is returned: ‘OK: <message-id>’ or ‘FAILED: <message reason>.’

Exceptions:

InvalidRecipient : Raised if the recipient is not valid.
InvalidAttachment : Raised if the file cannot be read, and hence attached to the message.

Send an Email

```
sendEmail(recipient:String,sender:String,title:String,
          body:String, attachment:File): String
```

Sends an Email to the recipient (an email address).

Parameters:

recipient : The email address that will receive the message.
sender : The sender address of the message.
title : The subject of the email.
body : The textual content of the message.
attachment : The file reference (a URL) to include in the message.

Outputs:

A string status is returned: ‘OK: <message-id>’ or ‘FAILED: <message reason>.’

Exceptions:

InvalidRecipient : Raised if the recipient is not valid.
InvalidAttachment: Raised if the file cannot be read, and hence attached to the message.

Send an Instant Message

```
sendIM(recipient: String, sender: String, body: String): String
```

Sends an instant message to a recipient (typically an email address). The content is passed in the body parameter. The sender parameter indicates the sender address.

Parameters:

recipient : The destination address that will receive the instant message.
sender : The address of the sender of the message.
body : The content of the instant message.

Outputs:

A string status is returned: 'OK: <message-id>' or 'FAILED: <message reason>'.

Exceptions:

InvalidRecipient : Raised if the recipient is not valid.
InvalidAttachment : Raised if the file cannot be read, and hence attached to the message.

Send an Voice Message

```
sendVoice(recipient: String, sender: String,
          body: String[0..1], audio: File[0..1]): String
```

Deposit a voice message to the recipient (a phone number). Body is used to pass a text to synthesize (text to speech) or alternatively audio is a File path reference to the audio file.

Parameters:

recipient : The destination phone number that will receive the voice message.
sender : An identifier for the sender of the message.
body (optional): The text to be synthesized.
audio (optional): A reference to a audio file to be played as the message.

Outputs:

A string status is returned: 'OK: <message-id>' or 'FAILED: <message reason>'.

Exceptions:

InvalidRecipient : Raised if the recipient is not valid.
InvalidAudioFile : Raised if the audio file cannot be read.

Send a message (generic)

```
send(recipient: String, sender: String, title: String, subject: String,
     body: String, attachment: File[0..1]): String
```

Send a message using a given media selected based on context information. The kind of message send depends on recipient format (a phone number? An email?). It may also depend on user preferences.

Parameters:

Recipient : The address that will receive the message.
sender : An address identifying the sender of the message.
title : The subject of the message.
body : The textual content of the message.
attachment : The file reference (a URL) of an attachment included in the message.

Outputs:

A string status is returned: 'OK: <message-id>' or 'FAILED: <message reason>'.

Exceptions:

InvalidRecipient: Raised if the recipient is not valid.
InvalidAttachment: Raised if the attachment file cannot be processed.

8.4 Short Messaging (SMS)

8.4.1 Overview

The SMS interface has been standardized by GSMA and is part of the OneAPI interfaces. This is the reformulation in SoaML.

See original specification at:

https://gsma.securespsite.com/access/Access%20API%20Wiki/SMS%20RESTful%20API.aspx

The SMS service enabler provides operations for:

- Sending an SMS to a terminal, for example from a Web page or desktop application.

- A polling mechanism for monitoring the delivery status of a sent SMS
- Being notified of delivery status
- A polling mechanism in order to receive SMS, for example a user sending text to a Web application from their mobile device.
- Being notified of SMS reception status (i.e. when an SMS is received by the Web application)

8.4.2 Interface Definitions

Three interfaces are defined. The SMS interface is the server side enabler. The SMSDeliveryNotification is a client side interface to allow a client application to receive notifications related to status of messages sent by the application to individuals. The SMSMessageNotification is another client side interface to allow receiving notifications of messages sent to an application.

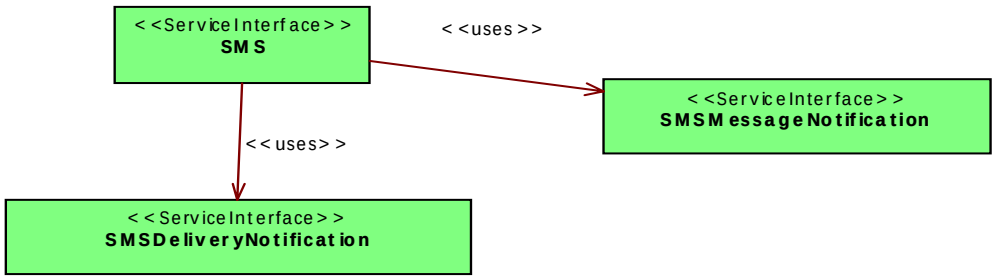


Figure 8.2 - Interfaces of the SMS Enabler

8.4.2.1 SMS Interface

The SMS interface is the server-side enabler interface.

Note: For readability reasons, optional multiplicities ([0..1]) in the arguments of service operations of this interface have been skipped in the diagram below. The details of signature can be found in the detailed service operation descriptions.

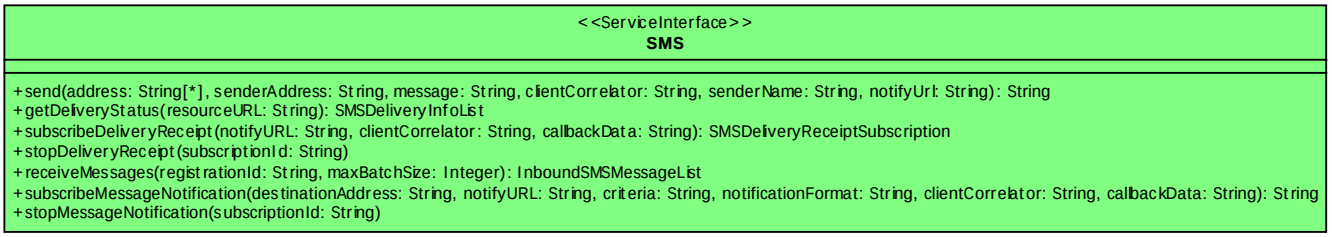


Figure 8.3 - SMS Interface

8.4.2.1.1 Data Types

The specification of the SMS interface involves the definition of seven data types and one enumeration: SMSDeliveryInfoList, SMSDeliveryInfo, SMSDeliveryStatus, SMSDeliveryReceiptDescription, SMSCallbackReference, InboundSMSMessageList, InboundSMSMessage.

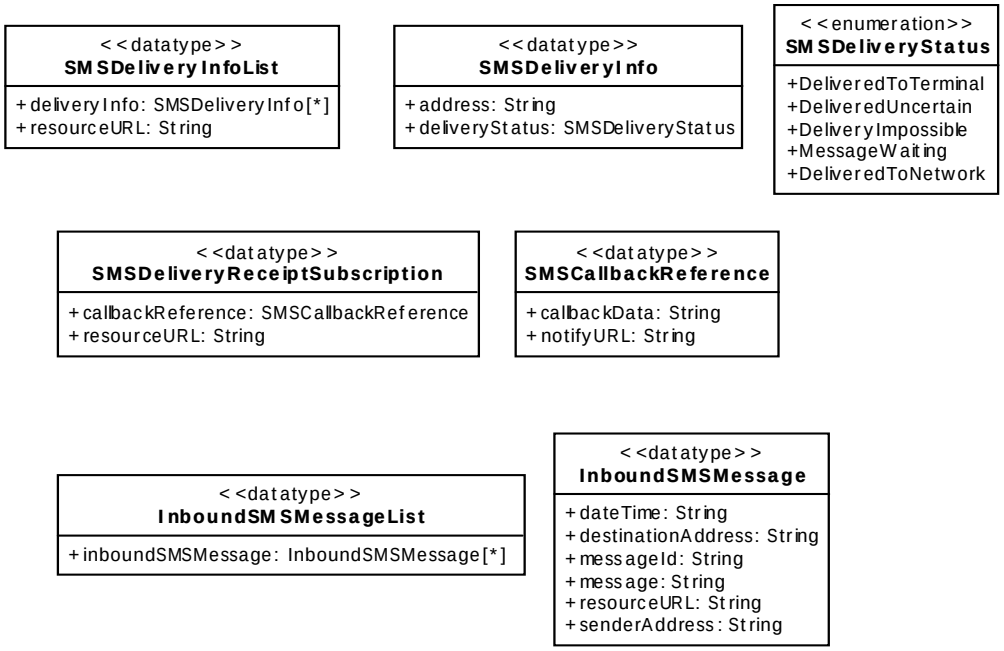


Figure 8.4 - Specific data types for the SMS interface

SMSDeliveryInfoList

A SMSDeliveryInfoList is used for receiving delivery notifications. It contains an URL to locate in the web the delivery information (*resourceURL* field) and the list of delivery reports, one per SMS address (*deliveryInfo* field).

SMSDeliveryInfo

A SMSDeliveryInfo contains the delivery information for a given address. It holds the address and the status value (deliveryStatus field).

SMSDeliveryStatus

The enumeration defining possible delivery status values: DeliveredToTerminal, DeliveredUncertain, DeliveredImpossible, MessageWaiting, DeliveredToNetwork.

SMSDeliveryReceiptDescription

A SMSDeliveryReceiptSubscription is used in notification subscriptions. It contains a *callbackReference* (information for treating the notification when received) and a URL to identify the subscription (*resourceURL* field).

SMSCallbackReference

A SMSCallbackReference contains information how to receive the subscribed notifications. The *callbackData* field contains the operation to call and the *notifyURL* field specifies the URL to be used by the SMS server for posting the notifications.

InboundSMSMessageList

An InboundSMSMessageList is used for retrieving SMS messages. It contains a list of InboundSMSMessages (inboundSMSMessage field).

InboundSMSMessage

An InboundSMSMessage contain all the details of a SMS message. This comprises: a *dateTime*, *destinationAddress*, *messageId*, *message* (the body of the SMS message), *resourceURL* (a URL to locate the message as a resource in the web) and the original *senderAddress*.

Note: Additional semantics details of these data types are provided inline with the description of the operations where they are used.

8.4.2.1.2 Operations

Send an SMS

```
send(address: String[*], senderAddress: String, message: String,
     clientCorrelator: String, senderName: String, notifyUrl: String): String
```

Sends an SMS to one or more terminals, i.e. mobile devices or SMS-enabled laptops.

Parameters:

address (multi-valued): Contains the list of addresses to which the SMS will be sent.

senderAddress: The address to whom a responding SMS may be sent.

message: Contains the message to be sent. Messages over 160 characters may end up being sent as two or more messages by the operator.

clientCorrelator : A string that uniquely identifies this create SMS request. If there is a communication failure during the request, using the same clientCorrelator (when retrying the request) allows the operator to avoid sending the same SMS twice.

senderName : The name of the sender to appear on the terminal.

notifyURL : The URL to notify the application for delivery receipts.

Outputs:

A resource URL is returned to serve as unique identifier to request delivery notifications.

Exceptions:

InvalidAddress : All the addresses are invalid.
InvalidNotificationUrl : The passed notification URL cannot be reached.

Query the delivery status of an SMS

```
getDeliveryStatus(resourceURL: String): SMSDeliveryInfoList
```

Parameters:

resourceURL : The identifier of the message for which the delivery status is requested.

Outputs

The output is a SMS*DeliveryInfoList* object containing the delivery information for each address that the application asked to send the message to, in a SMS*DeliveryInfo* array comprising the address and a *deliveryStatus* value.

The *deliveryStatus* value may be one of:

- "DeliveredToTerminal": Successful delivery to Terminal.
- "DeliveryUncertain": Delivery status unknown: e.g. because it was handed off to another network.
- "DeliveryImpossible": Unsuccessful delivery; the message could not be delivered before it expired.
- "MessageWaiting": The message is still queued for delivery. This is a temporary state, pending transition to one of the preceding states.
- "DeliveredToNetwork": Successful delivery to the network enabler responsible for routing the MMS.

Exceptions

InvalidMessageIdentification : The resourceURL is unknown as an identifier of the message.

Subscribe to SMS delivery notifications

```
subscribeDeliveryReceipt(
   notifyURL: String, clientCorrelator: String,
   callbackData: String): SMSDeliveryReceiptSubscription
```

Parameters:

notifyURL : URL to be used by the server to send notifications.

clientCorrelator : Uniquely identifies this create subscription request. If there is a communication failure during the request, using the same clientCorrelator (when retrying the request) allows the operator to avoid creating a duplicate subscription.

callbackData : A function name or other data that you would like included when the notification is received.

Outputs

A SMSDeliveryReceiptSubscription object containing the subscription information and a identifier of the subscription that can be used later to unsubscribe.

Exceptions

InvalidNotificationURL : The notification URL cannot be reached.

Stop the subscription to delivery notifications

```
stopDeliveryReceipt(subscriptionId: String)
```

Parameters:

subscriptionId : The identifier of the subscription (obtained when making the subscription).

Outputs

None

Exceptions

InvalidSubscription: The subscription id is not known to the system.

Retrieve messages sent to an application

```
receiveMessages(
   registrationId: String,maxBatchSize: Integer): InboundSMSMessageList
```

Parameters:

- registrationId: An identified agreed with the service operator.
- maxBatchSize: The maximum number of messages to retrieve in this request

Outputs

An InboundSMSMessageList object containing a list of InboundSMSMessage objects (one per address) with the following contained fields:

- *dateTime :* The date the message was received.
- *destinationAddress* : The number associated with your service (for example an agreed short code).
- *messageId* : A server-generated message identifier.
- *message* : The textual content of the message.
- *resourceURL* : A link to the message (seen as a web resource).
- *senderAddress* : The MSISDN or Anonymous Customer Reference of the sender.

Exceptions

RetrievalFailed: Retrieval of message failed.

Subscribe to notifications of messages sent to an application

```
subscribeMessageNotification(
   destinationAddress: String, notifyURL: String,
   criteria: String[0..1], notificationFormat: String[0..1],
   clientCorrelator: String[0..1], callbackData: String[0..1]) : String
```

Parameters:

destinationAddress : The MSISDN, or code agreed with the operator, to which people may send an SMS to the application.

notifyURL : Address to which notifications will be sent.

criteria (optional): Case-insensitve text to match against the first word of the message, ignoring any leading whitespace. This allows you to reuse a short code among various applications, each of which can register their own subscription with different criteria.

notificationFormat (optional): The content type that notifications will be sent in (example JSON).

clientCorrelator (optional): Uniquely identifies this create subscription request. If there is a communication failure during the request, using the same clientCorrelator when retrying the request allows the operator to avoid creating a duplicate subscription.

callbackData (optional): A function name or other data to be included in the notification.

Outputs

Returns the identifier of the subscription (usable to stop the subscription).

Exceptions

DestinationAddressInvalid: The destination address does not match the one agreed with the operator.
InvalidNotificationURL: The notification URL cannot be accessed.

Stop the subscription to message notifications

```
stopMessageNotification(subscriptionId: String)
```

Parameters:

subscriptionId : String

Outputs

None

Exceptions

InvalidSubscription : The subscription id is not known to the system.

8.4.2.2 SMS Delivery Notification Interface

The SMSDeliveryNotification interface is called by the SMS enabler to notify a client application on events related to the delivery of a SMS previously sent.

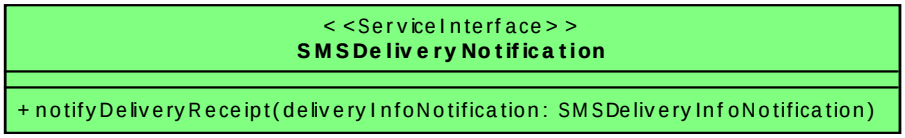


Figure 8.5 - SMSDeliveryNotification interface

8.4.2.2.1 Data Types

The specification of the SMSDeliveryInformation interface involves the definition of a specific data type SMSDeliveryInfoNotification, which in turn refers to the SMSDeliveryStatus enumeration (see definition data types of SMS interface).

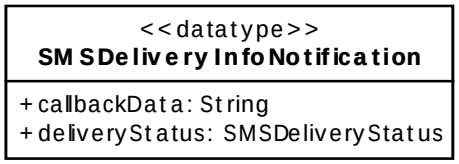


Figure 8.6 - Specific data types for the SMS Delivery Notification interface

SMSDeliveryInfoNotification

A *SMSDeliveryInfoNotification* contains the information passed in the notification. It includes a *callbackData* string and a *deliveryStatus*. The callbackData, typically the name of a function to call, is the data passed when subscribing to the notification (through the operation SMS::subscribeDeliveryReceipt). Possible values for the delivery status are: DeliveredToTerminal, DeliveredUncertain, DeliveryImpossible, MessageWaiting and DeliveredToNetwork (see SMSDeliveryStatus enumeration defined in “SMSDeliveryStatus” on page 12 and used by SMS::getDeliveryStatus).

8.4.2.2.2 Operations

Receive the notification of SMS delivery

`notifyDeliveryReceipt(deliveryInfoNotification: SMSDeliveryInfoNotification)`

This operation is called by the SMS enabler to notifiy the application when a SMS is delivered to a terminal or if delivery was impossible.

Parameters

deliveryInfoNotification: A SMSDeliveryInfoNotification containing the delivery information made of a callback information and a delivery status (see details in SMSDeliveryInfoNotification datatype description above).

Outputs

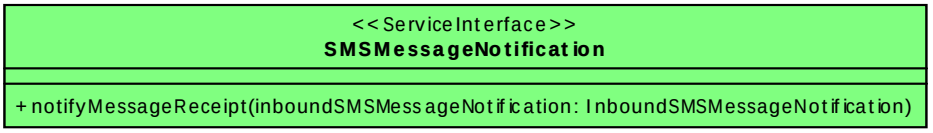
None

Exceptions

None

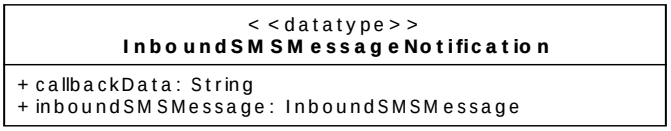
8.4.2.3 SMS Message Notification Interface

The SMSMessageNotification interface is called by the SMS enabler to notify an application the reception of SMS messagesfor which the application has subscribed.



8.4.2.3.1 Data Types

The specification of the SMSMessageNotification interface involves the definition of a specific data type: InboundSMSMessageNotification. This data type uses the InboundSMSMessage datatype defined in Data types of SMS interface (see “SMS Interface” on page 11).



Specific data types for the Inbound SMS Message Notification interface

- InboundSMSMessageNotification - An InboundSMSMessageNotification object contains the information passed in the notification. It includes a callbackData string and a InboundSMSMessage. The callbackData, typically the name of a function to call, is the data passed when subscribing to the notification (through the operation SMS::subscribeDeliveryReceipt). The InboundSMSMessage object contains all the information on the received message: a date time, the destination address, the identifier and the content of the message and the sender address (see additional details in the definition of operation SMS::receiveMessages).

8.4.2.3.2 Operations

Receiving the notification of arrival of messages for a client application

notifyMessageReceipt(inboundSMSMessageNotification: InboundSMSMessageNotification)

This operation is called by the SMS enabler to notify the application whether messages for him are available or not.

Parameters

inboundSMSMessageNotification: contains the information on the received message in a InboundSMSMessageNotification object which contains the callbackData and the details of the received SMS within a InboundSMSMessage object (see details in InboundSMSMessageNotification datatype description above).

Outputs

None

Exceptions

None

8.5 Multimedia Messaging (MMS)

8.5.1 Overview

The MMS interface allows an application to send and receive MMS messages. The MMS interface has been standardized by GSMA and is part of OneAPI interfaces. This is the reformulation in SoaML. Original specification can be found at:

https://gsma.securespsite.com/access/Access%20API%20Wiki/MMS%20RESTful%20API.aspx

The MMS service enabler provides operations for:

- Sending an MMS to a terminal, for example from a Web page or desktop application.
- Query the delivery status of an MMS.
- Subscribe to MMS delivery notifications and stop subscription.

- Retrieve a list of messages, with or without attachments.
- Subscribe to notifications of messages sent to your application and stop the subscription.

8.5.2 Interface Definitions

Provided below are three interfaces: the MMS interface is the "server-side" enabler interface and the two notification interfaces, which are both required client interfaces (to be implemented by applications in order to receive notifications from the enabler).

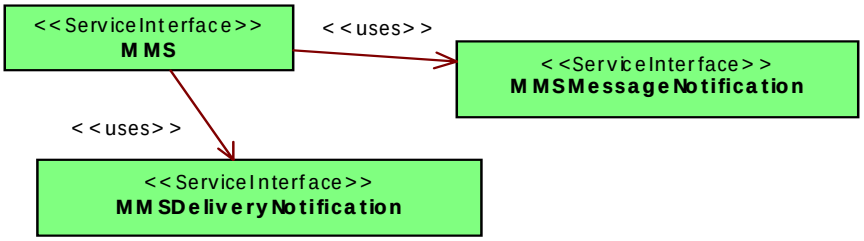


Figure 8.7 - Interfaces for the MMS Enabler

8.5.2.1 MMS interface

Note: For readability, optional multiplicities ([0..1]) in the arguments of service operations of this interface have been skipped in the diagram below. The details of signature can be found in the detailed service operation descriptions.

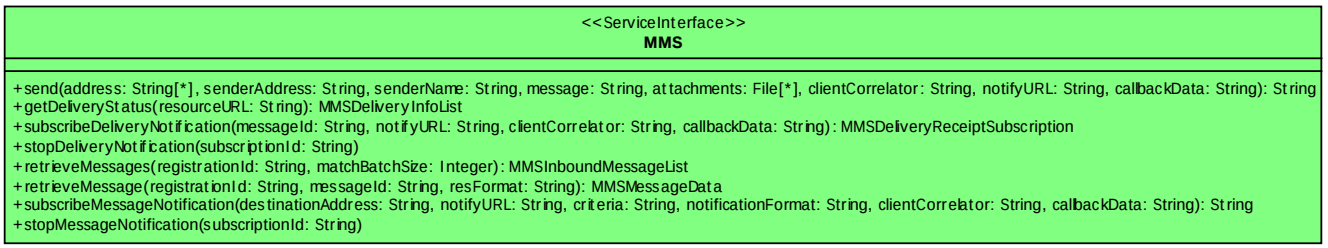


Figure 8.8 - MMS Interface

8.5.2.1.1 Data Types

The specification of the MMS interface involves the definition of eight data types and one enumeration: MMSDeliveryInfoList, MMSDeliveryInfo, MMSDeliveryStatus, MMSDeliveryReceiptDescription, MMSCallbackReference, MMSMessageData, MMSInboundMessageList, MMSInboundMessage.

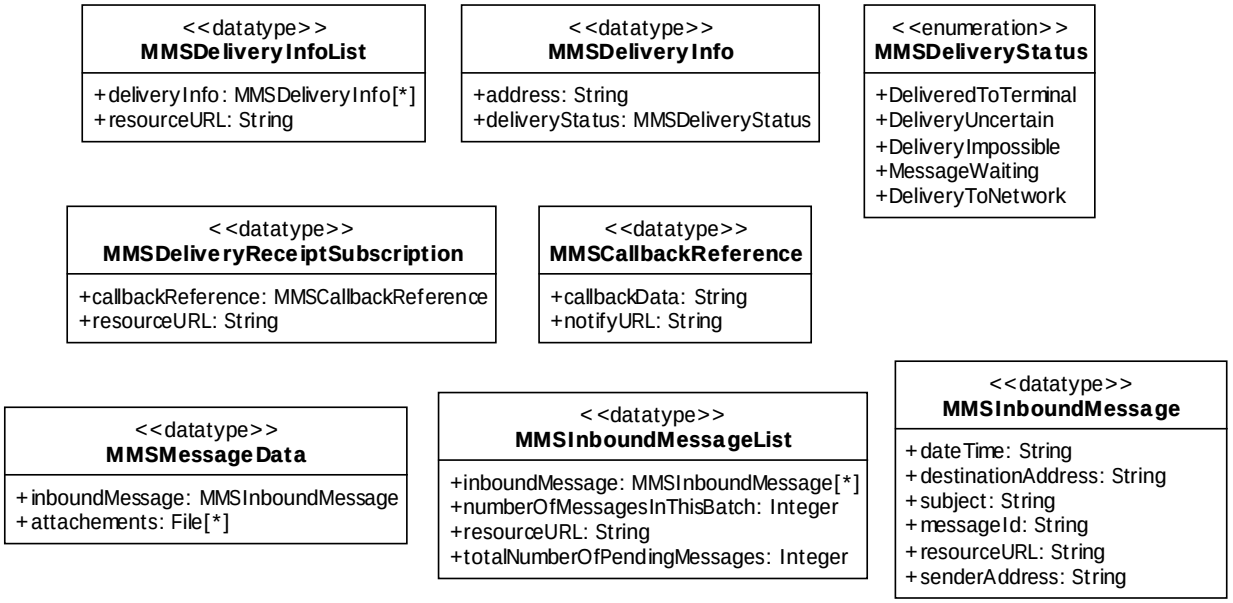


Figure 8.9 - Specific data types for the MMS interface

MMSDeliveryInfoList

A MMSDeliveryInfoList is used for receiving delivery notifications. It contains an URL to locate in the web the delivery information (*resourceURL* field) and the list of delivery reports, one per MMS address (*deliveryInfo* field).

MMSDeliveryInfo

A MMSDeliveryInfo contains the delivery information for a given address. It holds the address and the status value (deliveryStatus field).

MMSDeliveryStatus

The enumeration defining possible delivery status values: DeliveredToTerminal, DeliveredUncertain, DeliveredImpossible, MessageWaiting and DeliveredToNetwork.

MMSDeliveryReceiptDescription

A MMSDeliveryReceiptSubscription is used in notification subscriptions. It contains a *callbackReference* (information for treating the notification when received) and a URL to identify the subscription (*resourceURL* field).

MMSCallbackReference

A SMSCallbackReference contains information how to receive the subscribed notifications. The *callbackData* field contains the operation reference to call and the *notifyURL* field specifies the URL to be used by the MMS server for posting the notifications.

MMSInboundMessageList

An MMSInboundMessageList is used for retrieving MMS messages. It contains a list of MMSInboundMessages (inboundMessage field), the number of messages retrieved (*numberOfMessagesInThisBatch* field), the number of messages not yet retrieved (*totalNumberOfPendingMessages* field) and a URL to identify the message list as a web resource (*resourceURL* field).

MMSInboundMessage

An MMSInboundMessage contain all the details of a MMS message. This comprises: a *dateTime*, *destinationAddress*, a *subject*, *messageId*, *message* (the body of the MMS message), *resourceURL* (a URL to locate the message as a resource in the web) and the original *senderAddress*.

MMSMessageData

A MMSMessageData is used for retrieving the details of a given MMS message. It contains the data associated with the MMS (*inboundMessage* field of type MMSInboundMessage) and the list of attachment files (*attachments* field).

Note: Additional semantics details of these data types are provided inline with the description of the operations where they are used.

8.5.2.1.2 Operations

Send an MMS

```
send(address: String[*], senderAddress: String, senderName:
     String, message: String, attachments: File[*],
     clientCorrelator: String[0..1], notifyURL: String[0..1],
     callbackData: String[0..1]): String
```

Sends a Multimedia message (MMS) to a list of addresses.

Parameters:

address (multi-valued): Represents the phone recipients to reach. At least one *address* is required; in this case their MSISDN including the 'tel:' protocol identifier and the country code preceded by '+', for example tel: +16309700001. The address parameter also supports the Anonymous Customer Reference (ACR) if provided by the operator.

senderAddress : The address to whom a responding MMS may be sent.

senderName : The name to appear on the user's terminal as the sender of the message.

message : The textual content of the message.

attachments (multi-valued): List of file references containing the multimedia content (photo, video, text and so on).

clientCorrelator (optional): Uniquely identifies this create MMS request. If there is a communication failure during the request, using the same clientCorrelator (when retrying the request) allows the operator to avoid sending the same MMS twice.

*notifyUR*L (optional): The URL to which you would like a notification of delivery sent. The format of this notification is documented in the MMSDeliveryServiceNotification interface definition.

callbackData (optional): Any meaningful data the application would like send back in the notification, for example to identify the message or pass a function name, etc.

Outputs

Returns a resource URL that identifies uniquely the MMS sent. This identifier can be used to receive notifications on the status of the MSS.

Exceptions

InvalidAddresses: All addresses are invalid. The MMS cannot be sent.
InvalidNotificationURL: The notification URL cannot be reached.

Query the delivery status

```
getDeliveryStatus(resourceURL: String): MMSDeliveryInfoList
```

This operation allows an application to query the delivery status of a previously sent MMS.

Parameters

resourceURL : The identifier of the MMS for which the delivery status is requested.

Outputs

The output is a MMS*DeliveryInfoList* object containing the delivery information for each address that the application asked to send the message to in a MMS*DeliveryInfo* array comprising the address and a *deliveryStatus* value.

The *deliveryStatus* value may be one of:

- "DeliveredToTerminal": Successful delivery to Terminal.
- "DeliveryUncertain": Delivery status unknown: e.g., because it was handed off to another network.
- "DeliveryImpossible": Unsuccessful delivery; the message could not be delivered before it expired.
- "MessageWaiting": The message is still queued for delivery. This is a temporary state, pending transition to one of the preceding states.
- "DeliveredToNetwork": Successful delivery to the network enabler responsible for routing the MMS

Exceptions

InvalidMessageIdentification : The resourceURL is unknown as an identifier of the message.

Subscribe to MMS delivery notifications

```
subscribeDeliveryNotification(
  messageId: String, notifyURL: String, clientCorrelator: String[0..1],
  callbackData: String[0..1]): MMSDeliveryReceiptSubscription
```

This operation allows an application to subscribe to MMS delivery notifications.

Parameters

messageId : An identifier for identifying the MMS for which the subscription is requested.

notifyURL : The URL of the listener application used by the enabler to notify the subscribing client application.

clientCorrelator (optional): A string that uniquely identifies this create subscription request. If there is a communication failure during the request, using the same clientCorrelator (when retrying the request) allows the operator to avoid creating a duplicate subscription.

callbackData (optional): A function name or other data that the subscribing application would like included in the received notification.

Outputs

An MMSDeliveryReceiptSubscription object containing the subscription information and a identifier of the subscription which can be used later to unsubscribe.

Exceptions

InvalidMessageidentifier: The message identifier is invalid (because it is unknown or has expired).
InvalidNotificationURL: The notification URL cannot be reached.

Stop the subscription to delivery notifications

```
stopDeliveryNotification(subscriptionId: String)
```

This operation is used to stop the delivery of notification previously subscribed by the application.

Parameters

subscriptionId : The identifier of the subscription to stop.

Outputs

None

Exceptions

InvalidSubscriptionIdentification : The subscriptionId is unknown to the system.

Retrieve a list of messages sent

```
retrieveMessages(registrationId: String, matchBatchSize: Integer[0..1])
   : MMSInboundMessageList
```

This operation allows the calling application to retrieve MMS received by the server. Attachments are not included.

Parameters

registrationId : An identification negociated with the service operator. For instance ‘3456’ agreed with a OneAPI operator.

matchBatchSize (optional): The maximum number of messages to retrieve in this request.

Outputs

The output is an MMS*InboundMessageList* object containing:

- an MMS*InboundMessage* object with the following fields:
  - *dateTime :* The date the message was received.
  - *destinationAddress* : The number associated with your service (for example, an agreed short code).
  - *subject :* The *subject of* the message, which may determine whether you want to retrieve the entire MMS.
  - *messageId* : A server-generated message identifier.
  - *resourceURL* : A link to the message used to retrieve the entire message including attachments.
  - *senderAddress* : The MSISDN or Anonymous Customer Reference of the sender.
- the *numberOfMessagesInThisBatch*, which is the number of messages retrieved.
- a *resourceURL*, a self referring URL (the actual output seen as a web resource).
- the *totalNumberOfPendingMessages* awaiting retrieval from gateway storage.

Exceptions

RetrievalFailed: Retrieval of message failed.

Retrieve the full MMS including attachments

```
retrieveMessage(registrationId: String, messageId: String, resFormat: String)
  : MMSMessageData
```

This operation allows the application to retrieve an MMS received by the server including the attachments.

Parameters

registrationId : An identification agreed with the operator. For instance ‘3456’ agreed with anI operator.

resFormat : Indicates the format to retrieve the information. For example 'JSON' value ensures a JSON response content-type.

Outputs

The output is a MMSMessageData consisting of an MMS*InboundMessage* instance (see details in previous operation 'retrieveMessages') and a list of file attachments.

Exceptions

RetrievalFailed: Retrieval of message failed
InvalidMessageId: Message identifier unknown to the system.

Subscribe to notifications of messages sent to your application

```
subscribeMessageNotification(
   destinationAddress: String, notifyURL: String, criteria: String[0..1],
   notificationFormat: String[0..1], clientCorrelator: String[0..1],
   callbackData: String[0..1]): String
```

Parameters

destinationAddress : The MSISDN, or code agreed with the operator, to which people may send an MMS to the calling application.

notifyURL : The address to which notifications will be sent.

criteria (optional): Case-insensitve text to match against the first word of the message, ignoring any leading whitespace. This allows you to reuse a short code among various applications, each of which can register their own subscription with different *criteria*.

notificationFormat (optional) : Content type that notifications will be sent in – for instance JSON.

clientCorrelator (optional): A string that uniquely identifies this create subscription request. If there is a communication failure during the request, using the same clientCorrelator when retrying the request allows the operator to avoid creating a duplicate subscription.

callbackData (optional): A function name or other data that you would like included when the POST is received.

Outputs

The output is a subscriber identifier.

Exceptions

InvalidNotificationURL: The notification URL cannot be reached.

Stop the subscription to message notifications

`stopMessageNotification(subscriptionId: String)`

Parameters

subscriptionId : The subscription identifier returned by the prior subscription (which is being canceled).

Outputs

None.

Exceptions

InvalidSubscriptionIdentification : The subscriptionId is unknown to the system.

8.5.2.2 MMS Delivery Notification interface

The MMSDeliveryNotification interface needs to be implemented by a client to receive message delivery notifications.

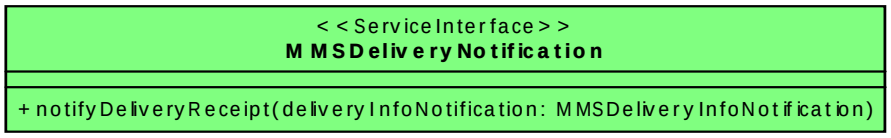


Figure 8.10 - Definition of MMSDeliveryNotification client interface

8.5.2.2.1 Data Types

The specification of the MMSDeliveryInformation interface involves the definition of a specific data type MMSDeliveryInfoNotification, which in turn refers to the MMSDeliveryStatus enumeration (firstly defined as a data type of the MMS interface (see “MMS interface” on page 19).

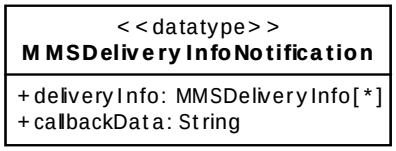


Figure 8.11 - Specific data types for the MMS Delivery Notification

MMSDeliveryInfoNotification

An MMSDeliveryInfoNotification object contains the information passed in the notification. It includes a callbackData string and delivery information for each destination address of the MMS by means of MMSDeliveryInfo objects. The callbackData, typically the name of a function to call, is the data passed when subscribing to the notification (through the operation SMS::subscribeDeliveryReceipt). The MSSDeliveryInfo (primarily defined in the context of the SMS interface) is a pair consisting of an address and a delivery status.

Possible values for the delivery status are: DeliveredToTerminal, DeliveredUncertain, DeliveryImpossible, MessageWaiting and DeliveredToNetwork (see MMSDeliveryStatus enumeration used by MMS::getDeliveryStatus).

8.5.2.2.2 Operations

Notifying delivery receipt at application side

notifyDeliveryReceipt(deliveryInfoNotification: MMSDeliveryInfoNotification)

This operation is invoked by the MMS enabler to pass notification information to a client application.

Parameters

deliveryInfoNotification: Contains all the notification information sent by the MMS server facility to the client application. It is structured as follow:

- A deliveryInfo array contains an address and a deliveryStatus for each user destination address of the MMS.

  The deliveryStatus can take same values than those used to query delivery status of an MMS, except for 'MessageWaiting', since that is the initial status. The callbackData string is also passed back to the client application, echoing what was provided when the message was sent or subscribed to delivery notifications.

Outputs

None.

Exceptions

None

8.5.2.3 MMS Message Notification interface

The MMSMessageNotification needs to be implemented by a client to receive message notifications.

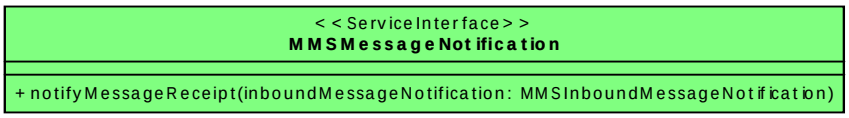


Figure 8.12 - Definition of the MMSMessageNotification

8.5.2.3.1 Data Types

The specification of the MMSMessageNotification interface involves the definition of a specific data type: InboundSMSMessageNotification. This data type implies in turn the definition of InboundMMSMessageNotification data type.

The MMSMessageNotification interface uses an MMSInboundMessageNotification data type which in turn uses a InboundMMSMessageNotificationdata type.

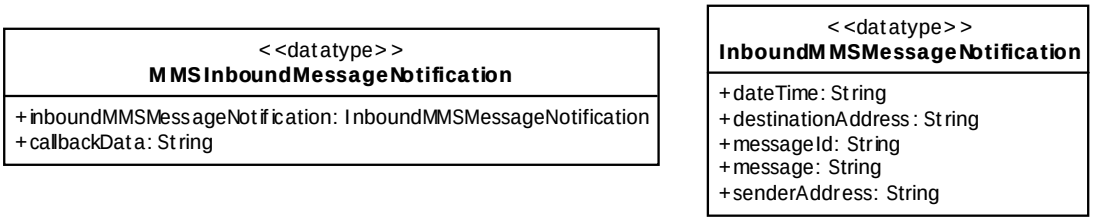


Figure 8.13 - Specific data types for the MMS Message Notification interface

MMSInboundMessageNotification

An MMSInboundMessageNotification object contains the information passed in the notification. It includes a callbackData string and an InboundMMSMessageNotification. The callbackData, typically the name of a function to call, is the data passed when subscribing to the notification (through the operation MMS::subscribeDeliveryReceipt).

InboundMMSMessageNotification

The InboundMMSMessageNotification object contains all the information on the received message: a date time, the destination address, the identifier and the content of the message and the sender address.

8.5.2.3.2 Operations

Notifying message receipt at application side

notifyMessageReceipt(inboundMessageNotification: InboundMessageNotification)

Parameters

inboundMessageNotification: The structure sent by the MMS enabler for every MMS received (matching the optional criteria if provided). The inboundMessageNotification object includes any callbackData for use by the calling application, and an inboundMMSMessage structure with the details below:

- the dateTime that the message was received,
- destinationAddress is the number associated with the calling application's service,
- messageId is a server-generated message identifier message is the MMS message itself
- resourceURL is a link to the message
- senderAddress is the MSISDN or Anonymous Customer Reference of the sender.

Outputs

None.

Exceptions

None

8.6 Click To Call

8.6.1 Overview

The click-to-call service allows an application to establish calls between two telephones. Since there is a notification mechanism, two interfaces are defined as: a server side ClickToCall and a client side ClickToCallNotification. The latter is the service that needs to be implemented by a client of the ClickToCall service in order to receive the notification. Notice that generally the provider of the ClickToCall imposes the implementation technology for receiving the notification - typically it will send an HTTP "GET" request with appropriate parameters.

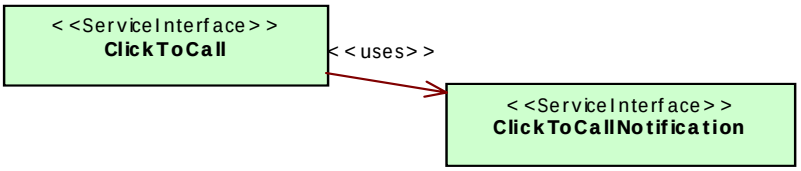


Figure 8.14 - Interfaces for the ClickToCall enabler

Note: This interface is a reformulation in UML of Orange Partner click-to-call API implementation (see http://api.orange.com/en/api/click-to-call-api/documentation).

8.6.2 Interface Definitions

8.6.2.1 Click To Call Interface

Note: For readability, optional multiplicities ([0..1]) in the arguments of service operations of this interface have been skipped in the diagram below. The details of signature can be found in the detailed service operation descriptions.

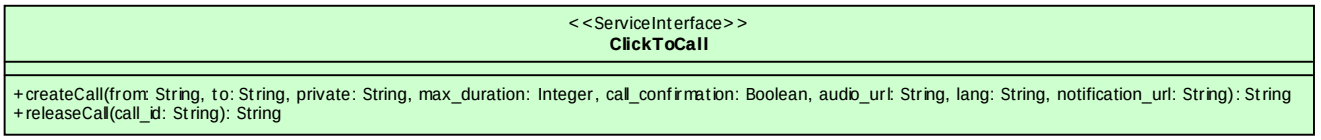


Figure 8.15 - Click to call service interface

This interface does not require the definition of specific data types (only primitive types are used).

8.6.2.1.1 Operations

Establishing a call

```
createCall(from:String,to:String,private:String[0..1],
           max_duration:String[0..1],call_confirmation:String[0..1],
           audio_url:String[0..1],notification_url:String[0..1]) : String
```

This is the server-side operation used by an application to establish calls and release calls between two phone numbers. A notification URL can provided to allow the enabler to send notifications regarding call status.

Parameters:

from : The phone number of the caller.

to : The phone number of the callee.

private (optional) : Indicates weather the phone numbers should be hidden. Possible values are ‘true,’ ‘false,’ ‘caller,’ and ‘callee.’

*max*_duration (optional): Max duration of the call in seconds. May be restricted by proper limitations of the enabler.

*call*_confirmation (optional): Indicates whether the system should ask the caller for a confirmation.

lang (optional): The language for the confirmation message if any (examples: FR, EN,...).

audio_url (optional) The audio file to be read for the confirmation message (if confirmation activated) or during the waiting period before establishing the call (confirmation not activated).
Note: The supported audio formats depends on the platform, hence they are not specified here.

notification_url (optional) Indicates the url for receiving the notification.

Outputs:

A call identifier of the new created call is returned. This identifier can be user to release the call.

Exceptions:

ForbiddenNumber: Either the caller or the callee number is forbidden or malformed.
MissingNumber: Missing ‘number’ parameter

Releasing a call

`releaseCall(call_id:String): String`

Parameters:

call_id : identifier of the call to be terminated.

Outputs:

The call identifier of the terminated call is returned (the one passed as parameter).
This identifier can be used to release the call.

Exceptions:

InvalidCallIdentifier: The call identifier is not valid (possibly because the call terminated).

8.6.2.2 Click To Call Notification Interface

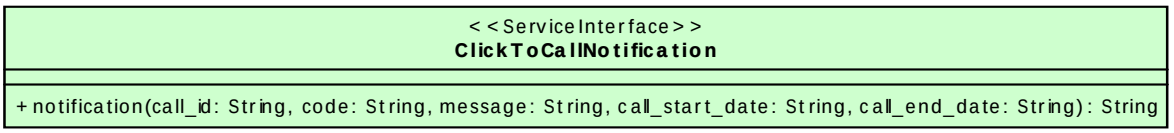


Figure 8.16 - Click to call notification interface

8.6.2.2.1 Operations

Receiving call status notification

```
notification(call_id: String, code: String, message: String,
             call_start_date: String, call_end_date: String): String
```

This operation is invoked by the ClickToCall enabler to notify a client on the status of a call.

Parameters:

call_id : Id of the call (the ID returned by createCall).

code: Status of the call (see the code table below).

message : Message of the event associated to code (see the code table below).

call_start_date : Start date of the call (in ISO 8601 date format). Not sent if the call is not fully established.

call_end_date : End date of the call (in ISO 8601 date format). Not sent if the call is not completed.

We provide below the values for code and corresponding message:

- 110 Caller is ringing
- 120 Caller has answered
- 130 Callee is ringing
- 200 Call is established
- 310 Caller hung up
- 330 Callee hung up
- 350 API hung up (max duration reached)
- 410 Caller is busy
- 420 Callee is busy
- 430 Caller is unavailable
- 431 Caller rejected call
- 432 Caller cancelled during negotiation
- 440 Callee is unavailable
- 441 Callee rejected call
- 442 Callee cancelled during negotiation
- 450 API cancelled call (max duration reached)
- 460 Caller did not confirm call
- 500 Internal error

Outputs:

The call_id value is returned as an acknowledgment.

Exceptions:

None

8.7 Location

8.7.1 Overview

This enabler provides the operations for localizing a person (typically based on GSM cell or on alternatives technologies like GPRS, UMTS and LTE).

This enabler typically works with some privacy management assumptions like the ability of the provider of the location service to identify and authenticate the sender of the requests. The data used to ensure privacy management – like a subscription id transparently passed at each request – is not exposed here but is typically part of the configuration accompanying the activation of the service.

Note: This interface is a reformulation in UML of OneAPI Location RESTful API. See original specification at:

https://gsma.securespsite.com/access/Access%20API%20Wiki/Location%20RESTful%20API.aspx

8.7.2 Interface Definitions

8.7.2.1 Location Service Interface

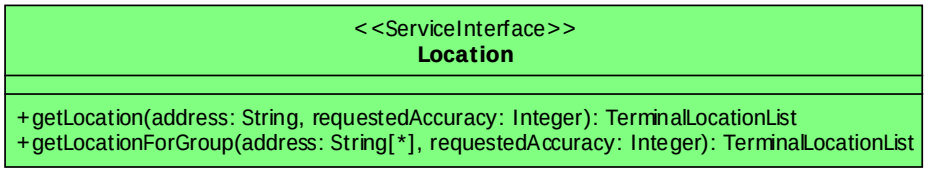


Figure 8.17 - Location interface and associated data

8.7.2.1.1 Data Types

The specification of the Location interface involves the definition of one specific data type TerminalLocationList, which in turn implies the definition of three additional data types: TerminalLocation, LocationInfo and LocationRetrievalStatus.

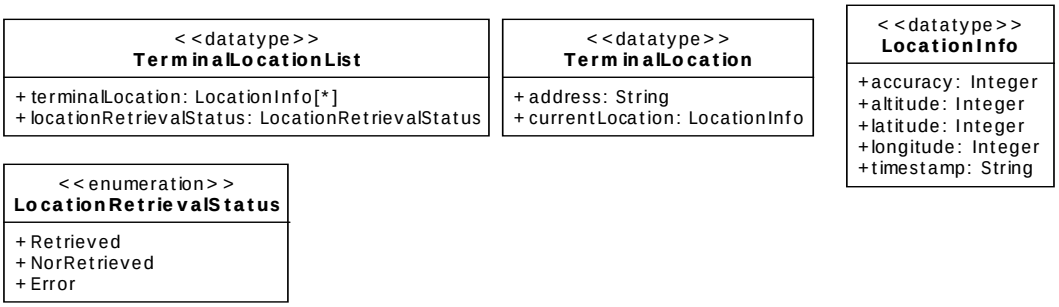


Figure 8.18 - Specific data types for the Location interface

TerminalLocationList

A TerminalLocationList contains a list of TerminalLocations (terminalLocation field) and the overall retrieval status (locationRetrievalStatus).

TerminalLocation

A TerminalLocation contains the address of the terminal (address) and the location information for the given address (currentLocation field).

LocationInfo

A LocationInfo gathers all the information concerning the location: accuracy, altitude, latitude, longitude and the timestamp of the location operation.

LocationRetrievalStatus

An enumeration representing three possible values regarding the tempative to retrieve the location: Retrieved, NotRetrieved or Error.

Note: Additional semantics details of these data types are provided inline in the description of Location operations where they are used.

8.7.2.1.2 Operations

Query the location of one mobile terminal

getLocation(address: String, requestedAccuracy: Integer): TerminalLocationList

Parameters:

address: The MSISDN or Anonymous Customer Reference of the mobile device to locate. The protocol and '+' identifier must be used for MSISDN.

requestedAccuracy: The preferred accuracy of the result, in metres. Typically, when an accurate location is requested it will take longer to retrieve than a coarse location. So requestedAccuracy=10 will take longer than requestedAccuracy=100.

Outputs:

The output is a TerminalLocationList object. It contains a unique TerminalLocation object, comprising:

- an "address" pair, to denote the terminal located, as per RFC 3966. Local and international numbers supported.
- a "currentLocation" object, comprising values for:
  - accuracy (result accuracy in metres)
  - altitude (metres)
  - latitude (Decimal Degrees, ISO 6709)
  - longitude (Decimal Degrees, ISO 6709)
  - timestamp (xsd:dateTime format)
- a "locationRetrievalStatus" pair, with possible values:
  - "Retrieved" (success)
  - "NotRetrieved" (unable to retrieve)

- “Error” (error retrieving location)

Exceptions:

- BadRequest : One of the parameters is malformed.
- AuthenticationFailure: The provided credentials are not valid.
- Forbidden: Authentification credentials not provided.
- ServerBusyAndServiceUnavailable. Server is temporarily not available.

Query the location of multiple mobile terminals

getLocationForGroup(address: String[*], requestedAccuracy: Integer) : TerminalLocationList

Parameters:

address (multi-valued) : The MSISDN or Anonymous Customer References of the mobile devices to locate. The protocol and ‘+’ identifier must be used for MSISDN.

requestedAccuracy : The preferred accuracy of the result, in metres. Typically, when an accurate location is requested it will take longer to retrieve than a coarse location. So requestedAccuracy=10 will take longer than requestedAccuracy=100.

Outputs:

The output is a TerminalLocationList object. It contains a TerminalLocation object for each passed address.

Exceptions:

- BadRequest : One of the parameters is malformed.
- AuthenticationFailure: The provided credentials are not valid.
- Forbidden: Authentification credentials not provided.

8.8 Synchronization

8.8.1 Overview

Synchronization Enabler role is to maintain user's data coherence between his services and devices. Two interfaces are needed. Firstly, on the server side the capabilities offered by the synchronization facility (Synchronization interface) and secondly the interface that synchronizable services need to implement in order to exploit the synchronization facility (Syncable interface). The diagram below shows the usage dependency between these two interfaces (Synchronization relies on availability of Syncable interfaces).

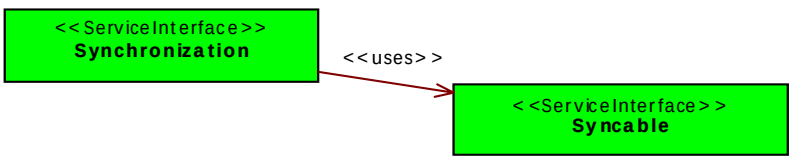


Figure 8.19 - Interfaces related with the Synchronization enabler

8.8.2 Context

This sub clause is non-normative. It is provided to allow implementers and users to understand the design of this interface.

The role of the Synchronization Enabler is to maintain user's data coherence between his services and devices. The most noticeable expected properties of a synchronization facility are:

- independence from the data to be synchronized
- ability to detect change on a data source
- ability to declare new data sources through a generic and decentralized interface

The OMA-DS standard describes precise protocols for realizing the synchronization on top of the HTTP infrastructure. An implementation of the synchronization enabler interface described here will typically be built on top of a OMA-DS compliant synchronization facility; however other supporting technologies could also be used.

Figure 8.20 in the left shows a data model representing the set of synchronized services/devices of a user. This data model illustrates the notion of Synchronization Sphere that represents a synchronized content (e.g., user's photos). A synchronization Sphere is composed of several data sources that are expressed as synchronization endpoints. Inner complexity is handled at the synchronization enabler level with the willingness to hide the complexity for end users and service developers. A Synchronization Endpoint Pattern represents synchronizable data source. When the system wants to add a new data source to a synchronization sphere, a synchronization endpoint is created taking a pattern as instantiation basis.

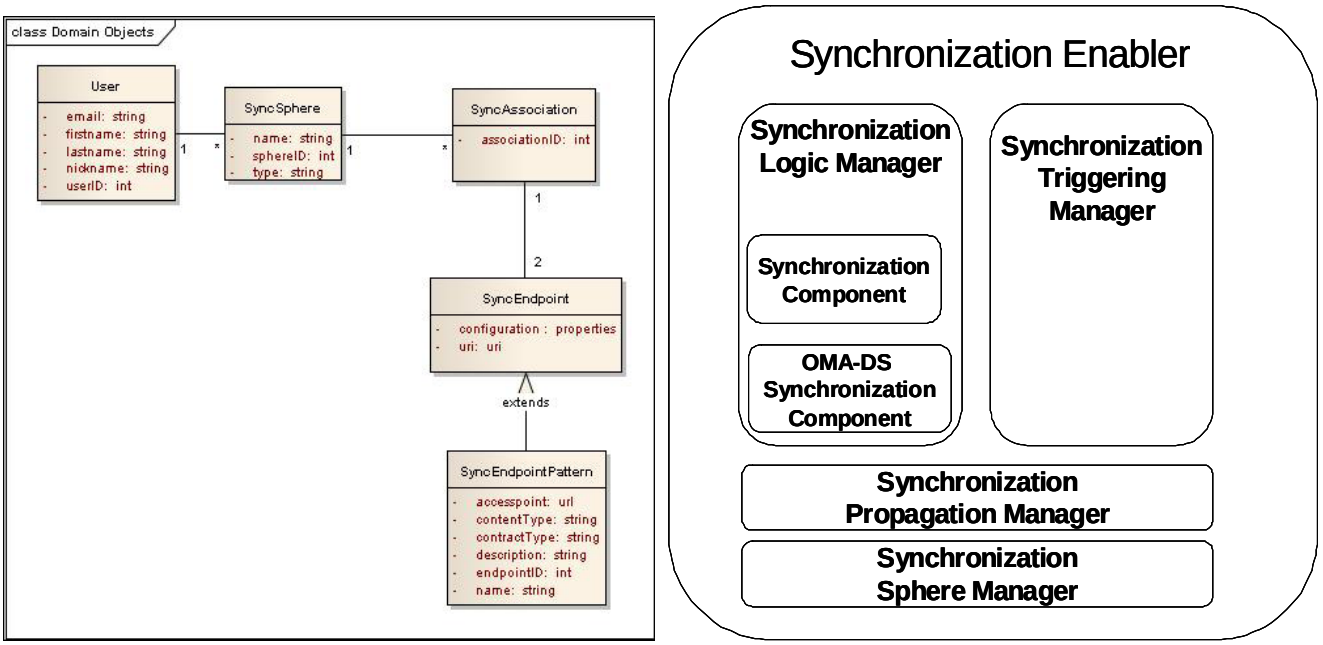


Figure 8.20 - Implementing the Synchronization Enabler: A data model and a possible internal architecture

As an enabler, the Synchronization Enabler exposes several interfaces:

- The first interface targets service developers and allows them to dig into Synchronization Enabler Data Model to display to users which services are part of their synchronization spheres.
- The second interface targets developers to allow users to extend their synchronization spheres with their services. This interface allow developers to enrich Synchronization Enabler scope by pushing new synchronizable data sources that can be reused by others actors.

Figure 8.20 on the right shows a typical architecture of the synchronization enabler in line with the data model. Implementers of the Synchronization enabler may select a different design. This is provided for helping understanding the underlying concepts.

In this architecture, we identify the following functional components:

- Synchronization Logic Manager (SLM) - This component is in charge of performing the actual synchronization process. To achieve this task the SLM relies on two major subcomponents:
  - The Synchronization Component: This component allows to provide connectors to new data sources (typically using REST).
  - The Synchronization Enabler provides also an OMA-DS layer that allows declaration of any OMA-DS compliant handset as element of a synchronization sphere.
- Synchronization Triggering Manager (STM) - this component is in charge of detecting if changes have occurred on a data source.
- Synchronization Propagation Manager (SPM) - Computing and ordering synchronizations to be performed are under the responsibility of the Synchronization Propagation Manager. When a new device or service is added to a user's synchronization sphere, the Synchronization Propagation Manager computes the new set of synchronization to be performed and how they shall be ordered. The idea is to put services that have the longer response time (mobile devices for instance) at the end of the synchronization chain.
- Synchronization Sphere Manager (SSM) - this component is in charge of:
  - Managing the topology of user's devices and services that are to be synchronized. The SSM allows to create Synchronization Spheres for a user and to define the services that are part of them.
  - Managing the triggering of synchronization session by external actors.

8.8.3 Interface Definitions

8.8.3.1 Synchronization Interface

The Synchronization Enabler covers several aspects like:

- access and configure user's synchronization spheres
- control the synchronization logic manager
- manage new data sources

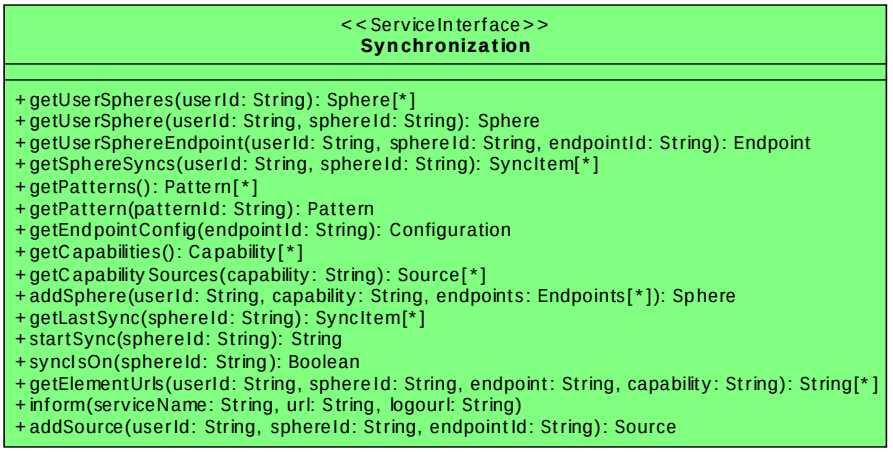


Figure 8.21 - Synchronization interface

8.8.3.1.1 Data Types

The specification of the Synchronization interface involves the definition of seven specific data types: Sphere, Endpoint, Capability, Source, Configuration, Pattern and SyncItem. The definition of Configuration in turn involves the definition of ConfigurationProperty.

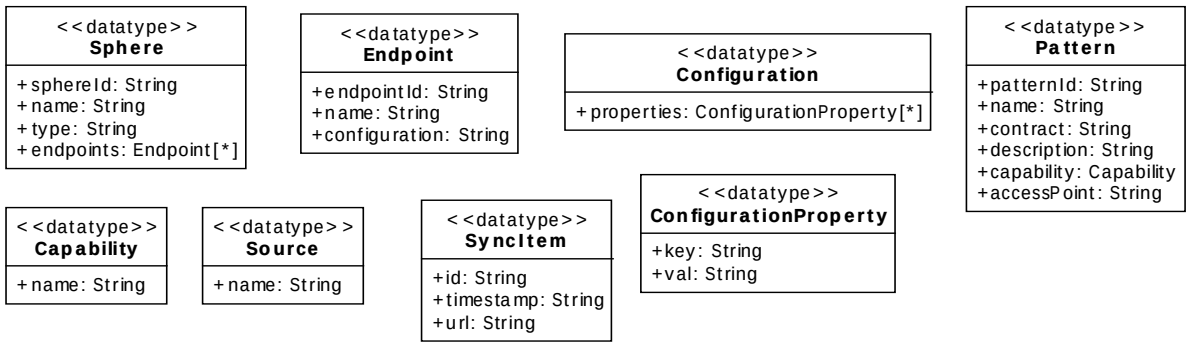


Figure 8.22 - Specific data types for the Synchronization interface

Sphere

A *Sphere* represents the sets of synchronized services and devices that are associated to a *type* of content (e.g. Photo content, Contacts, etc). The association between services is done by means of *endpoints*.

- *sphereId*: A unique identifier for the sphere within the synchronization framework.
- *name:* An "human readable" name for the sphere.
- *type*: The type of content of the sphere. Types of spheres are user-defined.

- *endpoints*: Represents connections to services providing a given data that can be synchronized.

Endpoint

An *Endpoint* represents one side of a synchronisation association between two services. An endpoint includes Configuration data that characterizes the service.

- *endpointId*: A unique identifier for the endpoint within the synchronization framework.
- *name*: A “human readable” name for the endpoint.
- *configuration*: The configuration data characterizing the connected service.

Configuration

A *Configuration*: represents the endpoint configuration as a whole. It includes a list of configuration properties characterizing the synchronized service.

- *properties*: list of configuration properties.

ConfigurationProperty

A *ConfigurationProperty* represents a given configuration property described with a key and a value.

- *key*: the key of the property
- *value*: the value of the property.

Pattern

A *Pattern* is a generic representation of a synchronizable data source. When the system wants to add a new data source to a synchronization sphere, a synchronization endpoint is created taking a pattern as instantiation basis. A pattern refers to a capability.

- *patternId*: A unique identifier for the pattern within the synchronization framework.
- *name*: An "human readable" name for the pattern
- *contract*: A text describing informally the conditions to fulfil to apply the pattern.
- *description*: A text providing a description of the pattern.
- *capability*: The capability supported by the pattern.

Capability

A *Capability* represents a given synchronization capability.

- *name*: The name identifying the capability.

Source

A *Source* represents an application managing some data that a user may want to synchronize. A capability can be supported by multiple sources.

- *name*: The name identifying the source.

SyncItem

A *SyncItem* represents a unit of synchronization. It is uniquely identified by a URL and has a timestamp.

- *id*: An identifier used by the synchronization system.
- *timestamp*: The time of the last synchronization.
- *url*: The URL of the unit of synchronization seen as a web resource.

8.8.3.1.2 Operations

Retrieving user's sphere list:

```
getUserSpheres(userId: String): Sphere[*]
```

This operation retrieves all spheres connected to a user.

Parameters
userId: The identifier of the user.

Outputs
The list of spheres.

Exceptions
UserInvalid: userId unknown to the system.

Retrieving sphere details:

```
getUserSphere(userId:String, sphereId: String): Sphere
```

Parameters

- *userId*: The identifier of the user.
- *sphereId*: The identifier of a Sphere.

Outputs
Returns a Sphere object containing all data associated to a sphere (including the list of endpoints)

Exceptions

- *UserInvalid*: userId unknown to the system.
- *SphereInvalid*: sphereId unknown to the system.

Retrieving endpoint details

```
getUserSphereEndpoint(userId:String, sphereId: String, endpointId: String): Endpoint
```

Parameters

- *userId*: The identifier of the user.
- *sphereId*: The identifier of a Sphere.
- *endpointId*: The identifier of an endpoint.

Outputs
The Endpoint details, including his configuration.

Exceptions

- *UserInvalid*: userId unknown to the system.
- *SphereInvalid*: sphereId unknown to the system.
- *EndpointInvalid*: endpointId unknown to the system.

Retrieving synchronizations attached to a sphere

```
getSphereSyncs(userId: String, sphereId: String): SyncItem[*]
```

Parameters

- *userId*: The identifier of the user.
- *sphereId*: The identifier of a Sphere.

Outputs
The output is the list of synchronization items of this sphere.

Exceptions

- *UserInvalid*: userId unknown to the system.
- *SphereInvalid*: sphereId unknown to the system.

Retrieving supported patterns

```
getPatterns(): Pattern[*]
```

Parameters
patternId: The identifier of the endpoint pattern.

Outputs
The output is the list of endpoint patterns supported by the synchronization enabler.

Exceptions
PatternInvalid: patternId unknown to the system.

Retrieving pattern details

```
getPattern(patternId: String): Pattern
```

Parameters
patternId: The identifier of the endpoint pattern.

Outputs
The output is the endpoint pattern definition.

Exceptions
PatternInvalid: patternId unknown to the system.

Retrieving endpoint configuration

```
getEndpointConfig(endpointId: String): Configuration
```

Parameters
: *endpointId*: The identifier of the endpoint.

Outputs
: The output is the configuration object containing all configuration properties of this endpoint.

Exceptions
: *EndpointInvalid*: endpointId unknown to the system.

Retrieving sync capabilities

```
getCapabilities(): Capability[*]
```

Parameters
: None

Outputs
: The output is the list of capabilities supported by the enabler.

Exceptions
: None

Retrieving list of sources supporting a capability

```
getCapabilitySources(capability: String): Source[*]
```

Parameters
: *capability*: The name identifying the capability.

Outputs
: The list of sources supporting the capability.

Exceptions
: *CapabilityNotSupported*: The capability is not supported by the enabler.

Adding a new sphere to a user

```
addSphere(userId: String, capability: String, endpoints: Endpoint[]) : Sphere
```

Parameters

- *userId*: The identifier of the user.
- *capability*: the name identifying the capability.
- *endpoints*: A list of endpoints objects.

Outputs
: A sphere object representing the new Sphere.

Exceptions

- *UserInvalid*: userId unknown to the system.
- *CapabilityNotSupported*: The capability is not supported by the enabler.
- *InvalidEndpoint*: One of the passed endpoints is invalid.

Retrieving last synchronization data for a sphere

```
getLastSync(sphereId: String): SyncItem[*]
```

Parameters
sphereId: The identifier of the sphere.

Outputs
The list of synchronization items (*SyncItems* objects) involved in the last synchronization.

Exceptions

- *SphereInvalid*: sphereId unknown to the system.
- *NotSynchronized*: No synchronization has been done for this sphere

Starting a synchronization

```
startSync(sphereId: String): String
```

Parameters
sphereId: The identifier of the sphere.

Outputs
Returns the sphereId if the synchronization start successfully, an empty string otherwise.

Exceptions

- *SphereInvalid*: sphereId unknown to the system.
- *SynchonizationCannotStart*: The synchronization could not start for any internal reason.

Retrieving synchronization process status

```
syncIsOn(sphereId:String): Boolean
```

Parameters
sphereId: The identifier of the sphere.

Outputs
A Boolean indicating if synchronization is in progress.

Exceptions
SphereInvalid: sphereId unknown to the system.

Retrieving item list urls for a data source

```
getElementUrls(userId: String, sphereId: String, endpointId: String,
               capability: String): String[*]
```

This operation gets the synchronization items of a data source, then collects and returns the URL fields.

Parameters

- *userId*: The identifier of the user.
- *sphereId*: The identifier of the sphere.
- *endpointId*:: The identifier of the endpoint.
- *capability*:: The name of the capability.

Outputs
: The list of URLs of all synchronization items of a data source

Exceptions

- *UserInvalid*: userId unknown to the system.
- *SphereInvalid*: sphereId unknown to the system.
- *EndpointIncalid*: endpointId unknown to the system

Informing the Synchronization Enabler that a new syncable source is available

```
inform(serviceName: String, url: String, logourl: String)
```

Parameters

- *serviceName* : Name of the service offering a SyncableInterface.
- *url*: The URL representing the service.
- *logourl*: The URL of the logo representing the service.

Outputs
: None

Exceptions
: *ServiceNotReachable*: The syncable interface cannot be accessed.

Creating/Adding a new syncable data source

```
addSource(userId: String, sphereId: String, endpointId: String): Source
```

Parameters

- *userId*: The identifier of the user.
- *sphereId*: The identifier of the sphere.
- *endpointId*:: The identifier of the endpoint.

Outputs

A source object representing the new source.

Exceptions

- *UserInvalid*: userId unknown to the system.
- *SphereInvalid*: sphereId unknown to the system.
- *EndpointIncalid*: endpointId unknown to the system.

8.8.3.2 Syncable Interface

The user's sphere can be enriched by declaring new data sources to be synchronized. This is done through the Syncable interface that a service may provide. This interface hides the complexity of implementation components (see Context sub-clause).

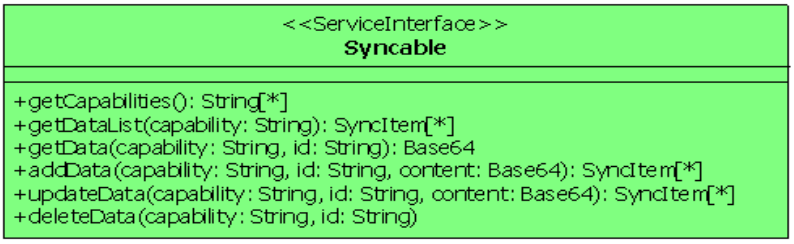


Figure 8.23 - Syncable interface

8.8.3.2.1 Data Types

The specification of the Syncable interface refers to the SyncItem data type (see definition in Synchronization interface) and involves the definition of a specific data type: Base64.

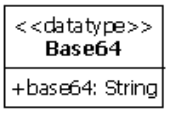


Figure 8.24 - Specific data types for the Syncable interface

Base64

A Base64 data type represents a text encoded using *base64* coding.

8.8.3.2.2 Operations

Retrieving service capabilities

```
getCapabilities(): String[*]
```

Parameters
None

Outputs
Returns the list of synchronization capabilities offered by the syncable service.

Exceptions
None

Retrieving item list for a capability

```
getDataList(capability: String): SyncItem[*]
```

Parameters
capability : The capability to filter the synchronization items (SyncItems).

Outputs
Returns the list of SyncItems that belong the the capability.

Exceptions
InvalidCapability : The capability is not known to the syncable service.

Retrieving item's data

```
getData(capability: String, id: String): Base64
```

Parameters

- *capability*: The capability of the synchronization item.
- *id*: The identifier of the synchronization item (which is unique for a given capability).

Outputs
The actual content of the item (for instance a JPEG image) encoded in base64.

Exceptions

- *SyncItemInvalid*: The synchronization item does not exists.
- *ItemContentInvalid*: The content of the item could not be accessed.

Adding a new Item

```
addData(capability: String, id: String, content: Base64): SyncItem[*]
```

Parameters

- *capability*: The capability of the synchronization item to add.
- *id*: The identifier of the synchronization item (which is unique for a given capability).
- *content*: The content of the synchronization item in base64 encoding.

Outputs
A list of synchronization items containing a unique SyncItem object representing the created item.

Exceptions
InvalidDuplicatedIdentifier: The identifier passed is already used.

Updating an item

```
updateData(capability: String, id: String, content: Base64): SyncItem[*]
```

Parameters

- *capability*: The capability of the synchronization item to add.
- *id*: The identifier of the synchronization item (which is unique for a given capability).
- *content*: The content of the synchronization item in base64 encoding.

Outputs
A list of synchronization items containing a unique SyncItem object representing the updated item.

Exceptions
SyncItemInvalid: The synchronization item does not exist.

Removing an item

```
deleteData(capability: String, id: String)
```

Parameters

- capability: The capability of the synchronization item to add.
- id: The identifier of the synchronization item (which is unique for a given capability).

Outputs
None

Exceptions

- *InvalidCapability* : The capability is unknown to the system.
- *InvalidItem*: The synchronization item is unknown.

8.9 Voice recognition and TTS

8.9.1 Overview

The available functions in the current version of the api allow any http client to pilot an Interactive Voice Response server (IVR), by requesting it to:

- Make or accept calls
- Say or play voice prompts
- Collect voice inputs (digit or speech recognition)
- Record

- Tear down (hang up or transfer)

8.9.2 Interface Definitions

8.9.2.1 Interactive Voice Response Facility Interface

The InteractiveVoiceResponseFacility service interface defines twelve simple operations exposing the basic functionality of a interactive voice response server. A specific data type named IVRResponse is defined in order to capture in a generic way the outputs of these operations. It includes a call identifier and a 'body' field to contain any response information.

Note: This interface is a reformulation in UML of Orange Partner MRF (Media Resource Facility) API implementation (see http://www.orangepartner.com/).

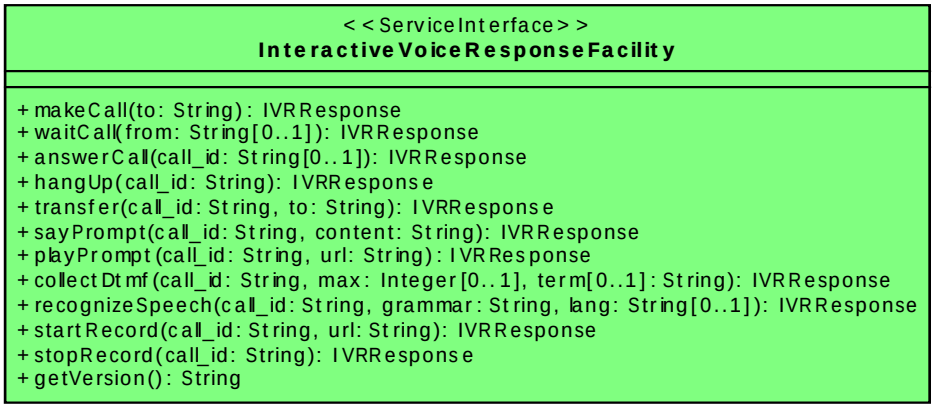


Figure 8.25 - IVR Facility service interface

8.9.2.1.1 Data Types

The specification of the IVRFacility interface involves the definition of one specific data type: the IVRResponse.

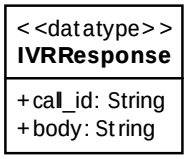


Figure 8.26 - Specific data types for the IVR Facility Interface

IVRResponse

IVRResponse is a generic data type defined to capture all responses of service operations of this enabler. It includes the following fields:

- *call_id*: A call identifier (unique in the system).
- *body*: Any additional information in the response (like the matched items when recognizing speech).

8.9.2.1.2 Operations

Making a call

`makeCall(to:String):IVRResponse`

This function requests the IVR to dial a phone number. It returns when the connection is established or if an error occurs.

Parameters:

to : A valid phone number

Outputs

Returns a IVRResponse object with 'call_id' assigned with the identifier for the new call. The 'body' field is left empty.

Exceptions:

- *Invalid_param*: Parameter ‘to’ is not a valid phone number.
- *Timeout*: the IVR could not fulfill the request within the time limit.
- *Hangup*: Callee rejected the call.
- *IVR_ERROR*: Any other error encountered by the IVR.

Waiting for a call

`waitCall(from:String[0..1]):IVRResponse`

This function requests the IVR to wait for a call for some duration (default value: 60s). It returns when the IVR receives an incoming call or when an error or timeout occurs.

Parameters:

from (optional) : A phone number. If provided the operation waits for a call to this number.

Outputs

Returns an IVRResponse object with ‘call_id’ of the incoming call. The ‘body’ field is left empty.

Exceptions:

- *Invalid_param:from*: Parameter from is not a valid phone number.
- *Timeout:* No call received before timeout.
- *IVR_ERROR*: Any other error encountered by the IVR.

Answering to a call

`answerCall(from:String[0..1]) : IVRResponse`

This function requests the IVR to answer to an incoming call. It returns when the connection is established or if an error occurs.

Parameters:

from (optional) : A phone number. If provided the operation answers a call to this number.

Outputs

Returns an IVRResponse object with ‘call_id’ of the answered call. The ‘body’ field left is empty.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Timeout* The IVR could not fulfill the request within the time limit.
- *Hangup* Caller hung up the call.
- *IVR_ERROR* Any other error encountered by the IVR.

Disconnecting a call

`hangUp(call_id:String):IVRResponse`

This function requests the IVR to hang-up the call identified by call_id. It returns when the connection has been closed or if an error occurs.

Parameters:

call_id : The call identifier.

Outputs

The output - and IVRResponse object, repeats the passed call identifier parameter.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Timeout* The IVR could not fulfil the request within the time limit.
- *IVR_ERROR* Any other error encountered by the IVR.

Transfering a call

`transfer(call_id:String,to:String): String`

This function requests the IVR to transfer the call identified by call_id to another phone number. It returns when the connection between the two users has been established or if an error occurs. Once transferred, no more requests to the IVR can be made on the original call_id.

Parameters:

- *call_id* : The call identifier.
- *to* : A valid phone number.

Outputs

Returns an IVRResponse object with ‘call_id’ of the new call. The ‘body’ field is left empty.

Exceptions:

- *Invalid_param:call_id* Parameter ‘call_id’ is not a valid call identifier.

- *Invalid_param:to* Parameter ‘to’ is not a valid phone number.
- *Timeout* The IVR could not fulfill the request within the time limit.
- *Transfer_failed* Transfer could not complete. For example the callee rejected the call.
- *IVR_ERROR* Any other error encountered by the IVR.

Playing a prompt message (text to speech)

`sayPrompt(call_id:String,content:String): IVRResponse`

This function requests the IVR to say a prompt to the user connected. It returns when the prompt has been played or if an error occurs.

Parameters:

- *call_id* : The call identifier.
- *content* : A URL encoded string representing the prompt.

Outputs
: Returns an IVRResponse object. The ‘body’ contains data regarding the selected voice. The exact content of body is left unspecified to accommodate to different implemented policies.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Timeout* The IVR could not fulfill the request within the time limit.
- *IVR_ERROR* Any other error encountered by the IVR.

Playing a prompt message from a file

`playPrompt(call_id:String,url:String): IVRResponse`

This function requests the IVR to play a file to the user connected. It returns when the IVR begins to read the file or if an error occurs.

Parameters:

- *call_id* : The call identifier.
- *url* : The URL of the file to read.

Outputs
: Returns an IVRResponse object. The field ‘call_id’ contains the passed call identifier.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Invalid_param:url* Parameter URL is not a valid URL.
- *File_error* The file cannot be red. Possible causes: file not found, wrong format.
- *Timeout* The IVR could not fulfill the request within the time limit.

- *IVR_ERROR* Any other error encountered by the IVR.

Retrieve DTMF input

`collectDtmf(call_id:String,max:Integer[0..1],term:String[0..1]) : IVRResponse`

This function requests the IVR to wait for specified number of digits from the connected user. It returns when max digits or terminal digit are received or if an error occurs.

Parameters:

- *call_id* : The call identifier.
- *max* (optional) : The maximum number of digits to retrieve from user input.
- *term* (optional): A terminal digit marking the end of the input.

Outputs

Returns an IVRResponse object. The field ‘call_id’ contains the passed call identifier and the field ‘body’ is a string consisting of recognized DTMF codes.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Invalid_param:max,term* Parameters max AND term are null.
- *Timeout* the IVR could not fulfill the request within the time limit.
- *IVR_ERROR* Any other error encountered by the IVR.

Retrieve and recognize speech input

`recognizeSpeech(call_id:String,grammar:String,lang:String[0..1]) : IVRResponse`

This function requests the IVR to start Speech Recognition with a list of given recognizable items. It returns when one of the items has been recognized or when an error or timeout occurs.

Parameters:

- *call_id* : The call identifier.
- *grammar* : The url-encoded list of recognizable items.

  Format: <item>item1</item><item>item2</item>…
- *lang* (optional) : Language for recognition. Default depends on the provider (example: fr-FR).

Output

Returns an IVRResponse object. The field ‘call_id’ contains the passed call identifier and the field ‘body’ is a string containing the recognized item.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Invalid_param:grammar* Parameter grammar is not valid.

- *Timeout* the IVR could not fulfill the request within the time limit.
- *IVR_ERROR* Any other error encountered by the IVR.

Start recording voice

`startRecord(call_id:String,url:String) : IVRResponse`

This function requests the IVR to start recording user's speech. It returns when the recording has started or if an error occurs.

Parameters:

- *call_id* : The call identifier.
- *url* : The location where the record will be stored.

Outputs
: Returns an IVRResponse object. The 'call_id' field copies the passed call identifier. The 'body' is a comma separated string containing the format used (for example 'wav' or 'mpeg') and information on rate quality.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Invalid_param:url* Parameter URL is not a valid URL.
- *Timeout* The IVR could not fulfill the request within the time limit.
- *IVR_ERROR* Any other error encountered by the IVR.

Stop recording voice

`stopRecord(call_id:String): IVRResponse`

This function requests the IVR to stop recording user's speech. It returns when the recording has been stopped or if an error occurs

Parameters:
: *call_id* : The call identifier.

Outputs
: Returns an IVRResponse object. The 'call_id' field repeats the passed call identifier. The 'body' field is empty.

Exceptions:

- *Invalid_param:call_id* Parameter call_id is not a valid call identifier.
- *Timeout* The IVR could not fulfill the request within the time limit.
- *IVR_ERROR* Any other error encountered by the IVR.

Retrieve the version of IVR facility

`getVersion() : String`

Returns the version of the IVR service.

Parameters:
None

Outputs
A comma separated string delivering information on the service. It contains in order the following information: a title, the service provider, and the version.

Exceptions:
None

8.10 Privacy

8.10.1 Overview

Two interfaces are defined here: the PrivacyResourceManager is used to validate a token to access a given functionality, and/or to access a set of attribute list. As an example the accessed functionality could be the user profile and the attribute list the list of fields such as phone number and address. On the other hand, the PrivacyPolicyManager provides CRUD operations (create/request,update,delete) to manage policy parameters in order to permit access to specific services. The detailed structure of policy is left opaque (Any datatype) to accommodate to different proprietary strategies.

8.10.2 Interface Definitions

Two complementary interfaces are defined here: PrivacyResourceManager and PrivacyPolicyManager.

8.10.2.1 Privacy Resource Manager Interface

Figure 8.27 - Privacy Resource Manager

8.10.2.1.1 Data Types

The specification of the PrivacyResourceManager interface involves the definition of two specific data types: AccessRequest and AttributeInfo. The later implies the definition of an ActionKind enumeration.

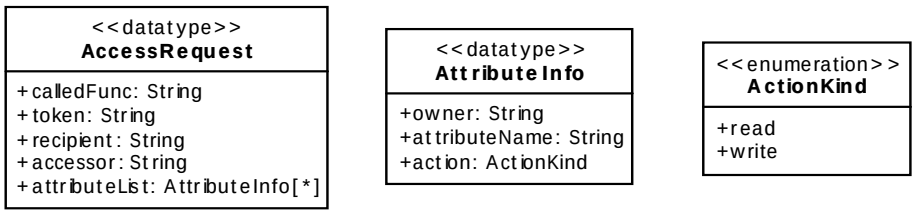


Figure 8.28 - Specific data types for the Privacy Resource Manager Interface

AccessRequest

The AccessRequest data type contains the information to control access to a protected resource. It includes:

- calledFunc: the function for which an access is requested.
- token: the token passed to be validated to validate an access request.
- recipient: an identity of the user that wishes to access the resource.
- accessor: an identity of the service being used to access the resource.
- attributeList : the list of attributes (AttributeInfo object) that will be accessed (for fine grained control).

AttributeInfo

The AttributeInfo data type contains the following fields:

- the owner of the attribute to be accessed (such as an Entity name or an Application name).
- attributeName: the name of the attribute.
- action: the action to be done (see ActtionKind).

ActionKind

The ActionKind enumeration defines the different kinds of actions that can be requested on an attribute. The two possible values are: read and write.

8.10.2.1.2 Operations

Token validation

validateToken(request: AccessRequest): AttributeInfo[*]

Return a list of personal attributes that can be accessed by the requestor with given token or null if the token is invalid. Within the AccessRequest parameter, the recipient field is the user identifier of the recipient who would like to access the attribute, the accessor field is an identifier of the service to be accessed. Each attribute in the attribute has a name, an owner (a user identifier) and permitted actions (read or write).

Parameters:

request: the request information provided by means of an AccessRequest object. This structure of a request is decribed above (see AccessRequest data type definition).

Outputs

The list of permitted attributes. The returned list may be a subset of the attributes requested in the request parameter. In that case authorization is partially provided. An empty list means that the access is not authorized at all.

Exceptions:

None

8.10.2.2 Privacy Policy Manager Interface

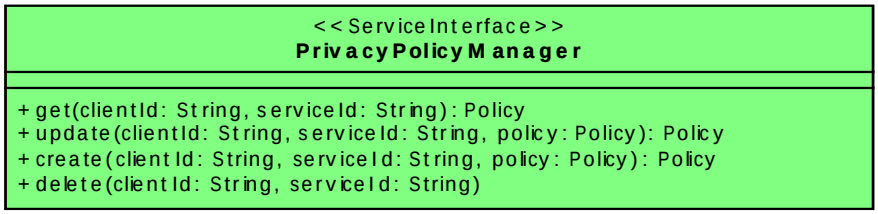


Figure 8.29 - PrivacyPolicyManager interface

8.10.2.2.1 Data Types

The specification of the PrivacyResourceManager interface involves the definition of one specific data type: Policy.

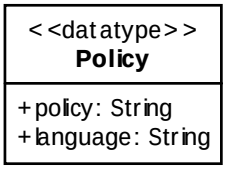


Figure 8.30 - Specific data types for the Privacy Policy Manager interface

Policy

The Policy data type contains the specification of the policy. The formalism used to express the policy is left user defined.

The Policy data type contains the following fields:

- policy: A string containing the specification of the policy.
- language: The formalism used to specify the policy.

8.10.2.2.2 Operations

Retrieve policy

get(clientId: String, serviceId: String): Policy

Returns the policy information for a given user and service.

Parameters:

clientId: The identifier of a registered user.
serviceId:. The identifier of a registered service.

Outputs

The Policy object for a given user an a given service.

Exceptions:

- InvalidClient : The passed clientId is unknown to the system.
- InvalidService : The passed serviceId is unknown to the system.

Update policy

update(clientId: String, serviceId: String, policy: Policy): Policy

Assign a new policy content for a given user and service.

Parameters:

- clientId: The identifier of a registered user.
- serviceId:. The identifier of a registered service.
- policy : The policy data to be stored in replacement of the existing one.

Outputs

The Policy object for a given user an a given service.

Exceptions:

- InvalidClient: The passed clientId is unknown to the system.
- IncvalidService: The passed serviceId is unknown to the system.
- InvalidPolicy: The policy passed is not correct.

Create policy

create(clientId: String, serviceId: String, policy: Policy): Policy

Adds a new policy for a given user and service.

Parameters:

- clientId: The identifier of a registered user.
- serviceId:. The identifier of a registered service.
- policy : The policy data to be stored.

Outputs

The Policy object for a given user an a given service.

Exceptions:

- InvalidClient: The passedclientId is unknown to the system.

- IncvalidService: The passed serviceId is unknown to the system.
- InvalidPolicy: The policy passed is not correct.

Delete policy

delete(clientId: String, serviceId: String)

Deletes an existing policy for a given client and a given service.

Parameters:

- clientId: The identifier of a registered user.
- serviceId:. The identifier of a registered service.

Outputs

None.

Exceptions:

- InvalidClient: The passedclientId is unknown to the system.
- InvalidService: The passed serviceId is unknown to the system.

9 TelcoML Composition Profile

9.1 Overview

This Clause is *normative*, with the exception of some sub-clauses explicitly marked as non-normative.

This clause specifies the “TelcoML Composition Profile” which, combined with the TelcoML Enabler Library form the TelcoML UML Profile. Technically it is defined as a UML profile specializing the SoaML UML profile.

The TelcoML Composition profile is intended to facilitate the development of composite telecommunication services by combining building blocks originating from the telecommunication industry (the so-called communication enablers) along with software components obtained from the internet. Typical services to be developed with TelcoML are mobile applications, which are portable across various smartphone platforms, or service logic deployed in the network by telecommunication service providers (CSP) or third party service providers. The TelcoML Enabler Library (see Clause 8) provides a set of standardized UML interfaces that can be used in service compositions when dealing with telecommunication facilities.

9.1.1 Relationship to SoaML

The TelcoML profile specialize SoaML profile and hence allowing telecom service designers to potentially take advantage of all the features of SoaML. Service logic in SoaML can be specified using various behavioral formalisms, for example, activity diagrams, state machines, and sequence diagrams. While this freedom is appropriate for addressing many needs related to SOA modeling, when designers are interested in interchanging executable specifications related to service compositions, it is necessary to make some restrictive choices. The TelcoML Composition profile selects state machines as the primary execution paradigm and defines a list of conventions to ensure some homogeneity in the way to represent service logic. State Machines have been selected because there is well known set of best practices using this formalism in the telecom community (inherited from ITU-T SDL experience) and also because it is very appropriate for representing interactive voice-based services.

When specifying service compositions using the TelcoML composition profile, the service designer need not represent SoaML participant components in order to define behavior. The behavior specification representing a service composition can be directly attached to the service operation owned by the service interface.

9.1.2 Relationship to Voice Profile

The Voice UML Profile defines means to express the design of voice based interactive applications based on a state-machine paradigm. All voice-oriented extensions defined in this profile have been re-introduced in the TelcoML profile to enable the definition of integrated services with multi-modal capacities. However TelcoML do not replace the Voice profile to deal with pure voice-interactive applications (because typical voice specifications are structured differently, where a *dialog* is the central structuring concept).

9.1.3 Relationship to CCXML and VoiceXML

The CCXML (Call Control eXtensible Markup Language) is designed to provide telephony call control support for dialog systems. It is connected to the VoiceXML standard. The TelcoML specification has no direct link with CCXML except that a CCXML implementation represents a natural candidate for supporting TelcoML state machines enhanced by voice and call control capacities.

9.2 Main concepts

Almost all concepts (except for Dialog) are specializations of the existing SoaML concept. However their usage in the TelcoML composition profile tends to be simpler since the focus is primarily the specification of composite services from an algorithmic point of view (rather than architectural point of view).

Technically, reuse of SoaML stereotypes - like ServiceInterface - is achieved by means of package import from the package defining the TelcoML Composition profile to the package defining SoaML. Instances of the imported stereotypes are not redefined by this profile.

9.2.1 Service Interface

A service interface in TelcoML (ServiceInterface concept from SoaML) consists basically of a list of service operations and, when relevant, the declaration of received and transmitted asynchronous events. The operations declare a signature with one or more input parameters and one output parameter. There are two possible styles to pass parameters and receive outputs: when message style is used (versus RPC style) the operation has a unique input parameter with a composite structure.

A usage dependency may be defined between a “server-side” service interface and a “client-side” service interface: this is relevant in telecom enablers that allow applications to receive asynchronous notifications from the enabler. In this case the specification of the "client" interface is part of the specification of the enabler. An example of this is given by the “Click to call” enabler in TelcoML Enabler Library.

In the case of a composite service there will be one operation representing the entry point of the orchestration with an explicit behavior attached. In the more general case, service operations may or may not expose a behavior specification.

ServiceInterface is represented in the TelcoML profile by the stereotype <<ServiceInterface>> on classes (reuse of SoaML stereotype).

9.2.1.1 Service Logic

By service logic we mean the specification of the behavior of a service operation. By convention, in the following “service logic” will be used as a shortcut of “logic of a service operation.”

The service logic may be “simple:” a sequence of actions with no parallelism and no asynchronous event reception, or “complex,” and in which case it is provided by a state machine. Conceptually, the first case can be treated as a “degenerate” case of the second case.

Use of the state machine paradigm potentially poses the possibility of representing arbitrary complex behaviors. Within a state machine we can refer to asynchronous events (accepting or transmitting events) which can be defined as part of the service interface.

The graphical rendering for state machines in TelcoML uses the *transition-oriented* notation specified by the UML specification (see UML2 Superstructure specification, Figure 15.44). When this notation is used the accepted events and actions to be executed when firing the transition are rendered by specific UML action icons.

Note: Tools providing no support or insufficient support on the transition-oriented notation can exploit activity diagrams to render state machines in transition-oriented manner. In TelcoML this is a notation variation point. However, TelcoML compliant tools exploiting this flexibility should have the ability to generate in XMI the same representation as if transition-oriented state machines were used.

ServiceLogic is represented in the TelcoML profile by the stereotype <<ServiceLogic>> of StateMachine.

Note: In TelcoML the notion of Participant (imported from SoaML) that exposes the internal structure of a component that realizes a service interface is out of focus. Service designers are free to provide such a view of the service by using full-fledged SoaML diagrams. However the interpretation of such view is not part of the TelcoML specification and a TelcoML execution tool (claiming the *TelcoML Execution* Conformance level) is not requested to support it.

9.2.1.2 Multi-modal Dialogs

In the telecommunication world services may be designed as muti-modal (for example accepting voice in combination with graphical interface inputs). TelcoML incorporates primitives of the Voice UML OMG standard in order to allow expressing interactions that imply voice recognition and synthesis. This is done by enhancing the list of actions and events that can be specified in state machine transitions, such as Play action, Inactivity, Utterance and Reject event types. In TelcoML dialogs are specific operations used to retrieve information from a user utilizing some form of interaction - for example voice or a graphical interface. These operations may be invoked explicitly from service logic.

9.3 Most commonly used stereotypes from SoaML

TelcoML adds a list of specific stereotypes. Potentially all stereotypes from SoaML can be used but the ones below play a distinctive role in TelcoML composition specifications.

- ServiceInterface: Defines the interface and responsibilities of a participant to provide or consume a service.
- Attachment: A part of a Message that is attached to rather than contained in the message.
- Property: Augments the standard UML Property with the ability to be distinguished as an identifying property
- MessageType: The specification of information exchanged between service consumers and providers.

9.3.1 Annotating service interfaces

TelcoML provides some basic mechanisms to classify different annotations that may enrich a model element involved in the definition of a service interface, for instance a service operation or a given parameter or the complete service interface. Three kinds of annotations are supported: annotations to provide semantic information (typically through links to pre-existing ontologies), annotation to provide non functional property information on the service and annotation to guide automatic generation of GUIs from the service specification. The details of these annotations are generally opaque (unspecified) to allow service designers use their preferred formalism. Also they serve as placeholders for future use of new formalisms.

9.3.2 Using graphical or textual notation

TelcoML defines a UML profile to allow service designer to take advantage of standardized UML conventions. However in some case it may be useful to use alternative textual notations. TelcoML which exploits state machines can take benefit of ALF textual notation, as described in Executable UML specification from OMG, to represent behavior.

9.4 TelcoML specific stereotypes

In this sub clause we provide the definition of specific TelcoML stereotypes.

9.4.1 Service logic related stereotypes

This section specifies the list of stereotypes dealing with service logic description.

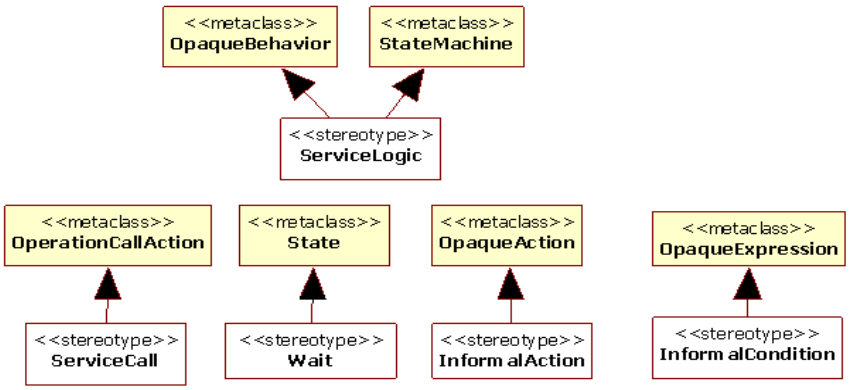


Figure 9.1 - Service Logic related concepts

In the following the semantics of the above stereotypes are defined and, when relevant, the specific notation is also covered. The information which can be derived from the provided diagram (such as the base metaclass) are not repeated.

9.4.1.1 ServiceLogic

A Service Logic defines the specification of the behavior of a service operation in the form of a state machine or an opaque definition. Both forms may coexist (useful in iterative development). A service logic is owned by a service operation.

Usage of <<ServiceLogic>> notation is not mandatory in a state machine diagram.

9.4.1.2 ServiceCall

A Service Call represents a call to a service operation within a service logic. From the point of view of the caller the call is always synchronous (it blocks until a response is received). However the effect of the call may be asynchronous. For example invoking “sendSMS” returns immediately but the SMS may take some time to arrive to the target phone.

Note: Calls to operations that are not service operations use the “regular” rectangle box representing actions in transition-oriented UML state machines.

A specific icon is used to notate a ServiceCall. It is similar to a “send event action,” except that the left side border has a triangle. Depending on tool presentation options and preferences, the call action text specification may be placed outside the box or inside the box. Also the <<ServiceCall>> stereotype indication may not appear in the icon. Another presentation option is use of the “regular” rectangle for action representation rather than the icon. However in such case the stereotype indication is mandatory. Figure 9.2 depicts various presentation options.

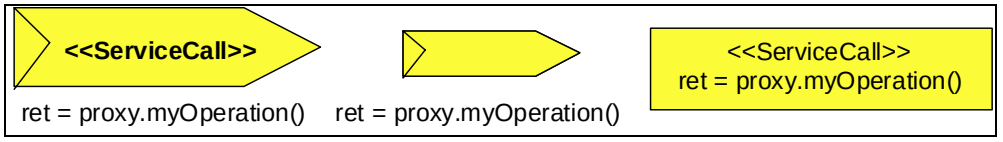


Figure 9.2 - Notations for service calls

9.4.1.3 Wait

A wait state is a state containing at least one outgoing transition triggered by the reception of an event.

A Wait state is notated as a regular state with <<Wait>> stereotype indication.

9.4.1.4 InformalCondition

An informal condition is a condition in a choice state specified informally, for instance by using natural language. An InformalCondition is a kind of OpaqueExpression..

An informal condition is notated as a regular Choice vertex with stereotype indication <<InformalCondition>>.

9.4.1.5 InformalAction

An informal action is an action definition specified informally, for instance by using natural language. An InformalAction is a kind of OpaqueAction.

An InformalAction is notated as a regular action (a rectangle box) with stereotype indication <<InformalAction>>.

9.4.2 Voice and multi-modal interaction related stereotypes

This sub clause specifies the list of stereotypes related to voice and other means of interaction.

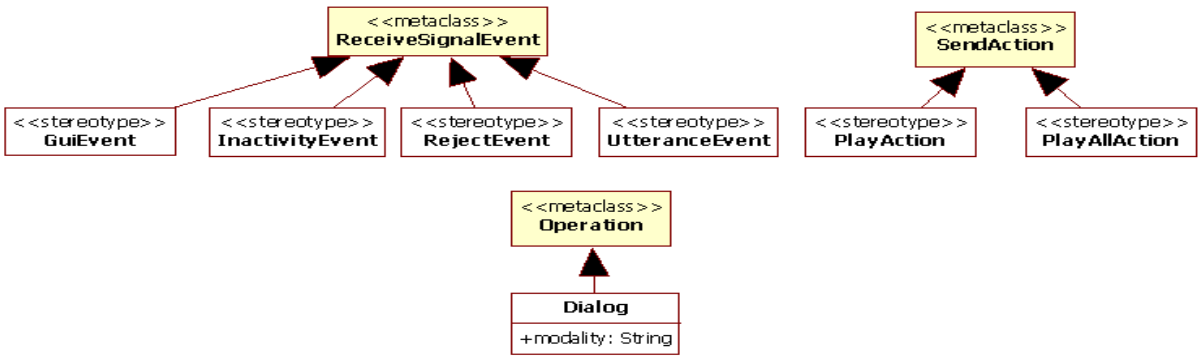


Figure 9.3 - Voice and multi-modality related concepts

In the sub clauses below we provide the semantics of the stereotypes and, when necessary the specific notation is also provided. The information which can be derived from the accompanying diagram is not repeated.

9.4.2.1 Dialog

A dialog represents an operation used to retrieve information from an end user, possibly using alternate methods (graphical interface, voice conversation, and so on).

Attributes

modality: String

Modality indicates the interaction mode. The attribute modality has two predefined values: “gui” and “voice.” Other values may be used.

The stereotype to be used is <<Dialog>>.

9.4.2.2 PlayAction

A play action represents the action of issuing an audio message to a user (text to speech or audio playing). This message can be interrupted by the user (bargin mode is active).

The stereotype to be used is <<Play>>.

9.4.2.3 PlayAllAction

A play all action represents the action of issuing an audio message to a user (text to speech or audio playing). This message cannot be interrupted by the user (bargin mode is active).

The stereotype to be used is <<PlayAll>>.

9.4.2.4 UtteranceEvent

An utterance event is an event definition representing the acceptance and recognition of voice input from the user.

The stereotype to be used is <<UtteranceEvent>>.

9.4.2.5 RejectEvent

A reject event is an event definition representing rejection - a recognition failure - of voice input from the user.

The stereotype to be used is <<RejectEvent>>.

9.4.2.6 InactivityEvent

A inactivity event is an event definition representing the absence of input from the user.

The stereotype to be used is <<InactivityEvent>>.

9.4.2.7 GuiEvent

A GUI event is an event generated by a graphical interface.

The stereotype to be used to denote this concept is <<GuiEvent>>.

9.4.3 Annotations for service interface elements

This sub clause defines specific concepts to annotate model elements used in service interface descriptions with some useful information: non functional properties, semantic annotations, and GUI annotations.

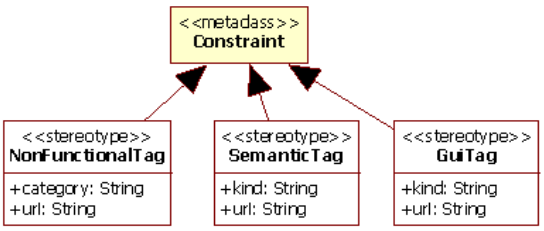


Figure 9.4 - Annotation facilities

In the following text we provide the semantics of the three stereotypes and, when relevant the specific notation. We do not repeat information that can be derived from the diagram.

9.4.3.1 SemanticTag

A semantic tag annotates an element within a service description (such as a service interface or a service operation) with some semantic information. Semantic annotations are typically realized by attaching references to concepts predefined in ontologies. Attaching semantic information to service descriptions is often useful for automatic service discovery.

Attributes

kind: String

Indicates to which kind the semantic annotation belongs. Pre-defined (normative) kinds are: goal, effect, precondition, postcondition and semantictype. Other non-normative values for the ‘kind’ attribute may be defined by the user.

A goal semantic annotation applies only to service interfaces or a given service operation.

preconditions and postconditions annotations apply to service operations.

An effect annotation apply to service operations: it can be used to represent side-effect semantics.

A semantictype annotation applies to a parameter. It typically references a concept in an ontology. Example: a ‘lang’ parameter may technically have String datatype but can be connected to the ‘Language’ concept defined by an ontology.

url: String

Gives the address of a concept defined in an ontology.

A Semantig tag is technically defined as a kind of Constraint. The value specification (inherited from Constraint metaclass) may be used to give details on preconditions and post-conditions.

A semantic tag is notated by a Note rectangle with <<SemanticTag>> stereotype indication.

9.4.3.2 NonFunctionalTag

A non functional tag annotates an element of a service description (such as interfaces or operations) with information concerning non functional features like Quality of Service, cost, availability and so on.

Attributes

category: String

Indicates to which category the non functional tag belongs.

The classification is user-defined, to accommodate to different preferences and future standardization.

url: String

Gives the address of a concept representing the non functional feature (typically as part of an ontology).

A non functional tag is technically defined as a kind of Constraint. The value specification (inherited from Constraint metaclass) may be used to give details on the described features (like cost conditions).

A non functional tag is notated by a Note rectangle with <<NonFunctionalTag>> stereotype indication.

9.4.3.3 GuiTag

A GUI tag represents information to guide GUI generation from the service interface definition. For example, a 'text' parameter may be annotated as "combo-box" to render it as a combo-box. The detailed classification ans structure is of these tags are left unspecified in TelcoML to allow service designer to adopt their own strategies based on the kind of GUI they want to generate.

Attributes

kind: String

Allows the classification of GUI information. This classification is user-defined.

url: String

Allows to refer to a GUI type (for instance a URL that identifies a widget type, such as a Combo-box in a pre-existing GUI library).

9.4.4 Additional presentation options

TelcoML notation defines two optional notational short-hands: one related to junctions pseudo-states and the other related to action sequences.

9.4.4.1 Junction pseudo-states

The regular UML notation for a Junction pseudo-state is an unnamed small black-filled circle. When there is a unique outgoing transition from a junction node, a TelcoML tool may propose the ability to assign a name to the junction pseudo-state: this is interpreted as a short-hand notation for a transition originating from the junction vertex to the state denoted by the junction name. The short-hand is only applicable if there is no name conflict.

This short-hand can be used to improve the readability of a service logic diagram (avoiding excessive amount of arrows in loops). The Figure below illustrates the use of this shorthand.

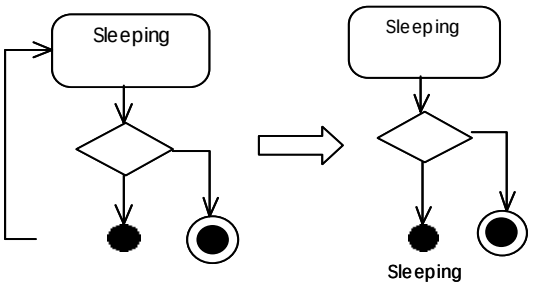


Figure 9.5 - Shorthand for junctions

9.4.4.2 Action sequences

State Machine UML notation allows using a single rectangle to denote an ordered sequence of actions. In complement to this TelcoML allows the following shorthand: the text content of the rectangle representing an action sequence may start with a slash symbol ('/'). In such case the rectangle is connected to a UML note containing the detail of the actions.

This short-hand can be used to limit the size of diagrams. An example of use of this notation is presented below.

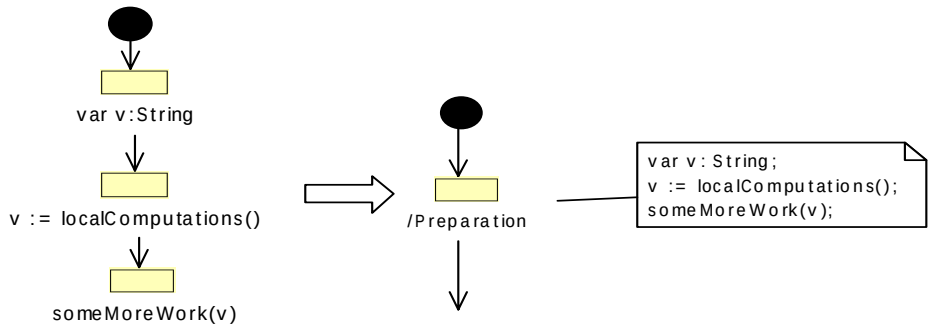


Figure 9.6 - Shorthand for Action Sequences

9.5 Examples

This sub clause is non-normative.

9.5.1 “Send by SMS weather in Paris translated in English”

The service “Send by SMS weather in Paris translated in English” is a very simple composite service that aggregates one telecom enabler (messaging) and two capabilities from internet (weather broadcast and translation). This example does not require managing asynchronous events, hence the state machine representing the behavior can be replaced by a simple sequence of actions.

First we provide the list of composed service interfaces and then the composite service itself. The SMS sent uses the “Generic messaging” capability from the TelcoML library, hence this capability is not explained here.

The two used interfaces (besides the Messaging service) are “Meteo France” and “Translator.”

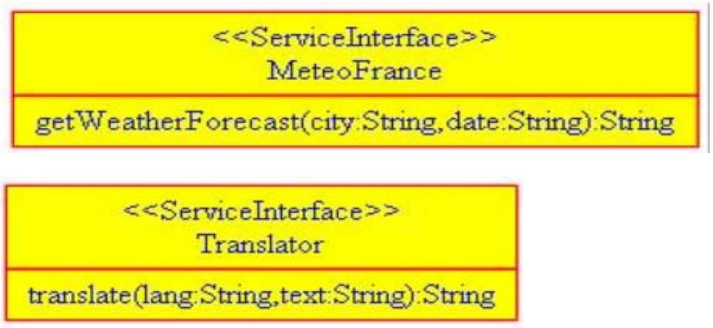


Figure 9.7 - Interface of the composed services

The interface of the composite service inquires the time of weather information (today, tomorrow, next week?) and also the telephone number destination where the weather summary will be sent.

Figure 9.8 - Interface of the composite service

Finally we specify the sequencing of service calls. In this case no intermediate computation is needed. The three service invocations can be done immediately after retrieving service proxies in an initialization step.

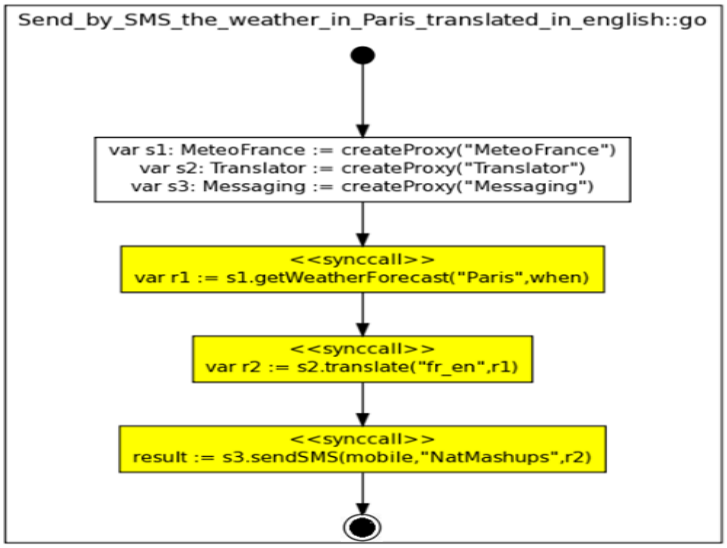


Figure 9.9 - Composite Service Logic

9.5.2 The Dinner Planning Example

We provide below an example of an integrated composite service that combines telco enablers and IT facilities. The composite service is designed using the TelcoML composition profile.

The E-tourism dinner planning scenario is as follow:

- An End User is on travel in a city. Because he does not want to waste time trying to find a good restaurant for his dinner he will delegate this task to a specialized dinner planning service. In the morning, he sends an SMS to the Service dinner planning requesting a search for "best recommended" restaurant at 20:00 near the location where he plans to be at that time, and respecting some criteria (such as the type of food).
- At dinner time (20:00), the Service locates suitable restaurants based on the end user geographic position.
- The Service sends a message to the End User containing the list of restaurants located in the surrounding area including the contact information for dinner reservation.
- The End User activates a call to the restaurant of choice using the restaurant contact point information.

The components that need to be in place for this scenario are:

- A Personal Agenda, to store from the user his willingness to be notified at dinner time.
- A Localization service, which will find the user's location based on GSM network information.
- A SMS or Instant Messaging enabler to notify the user when the list of restaurants is available.
- A Yellow Pages service to find the restaurants near the location of the user.
- A Third Party Call component to activate the call to the selected restaurant.

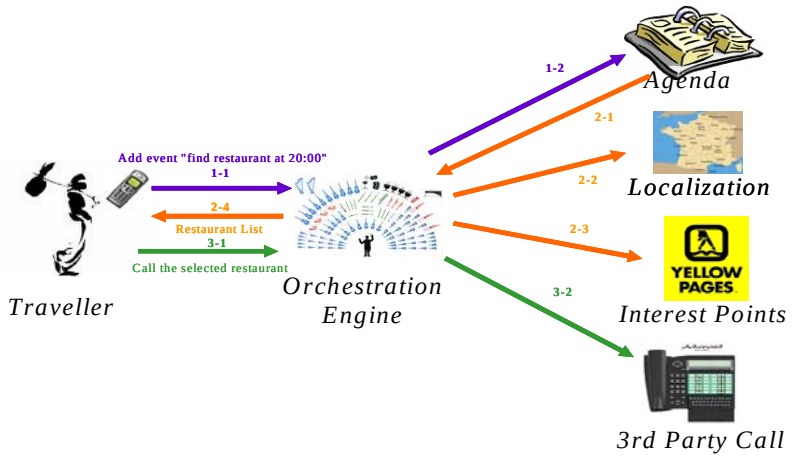


Figure 9.10 - Dinner planning scenario

From the point of view of the orchestrator, the scenario has three temporal phases:

- The orchestration engine receives the user request (1-1) and registers the event in the personal agenda (1-2).
- At dinner time, the orchestrator receives the reminder from the personal agenda (2.1) and subsequently invokes the localization services (2.2) to obtain the location information for the traveler. Then it requests the points of interest in the yellow pages services (2.3), collects the responses and sends the results to the traveler (2.4).

Finally, if the user selects a restaurant, the orchestrator receives the request (3.1) and invokes the 3rd party call service to establish the communication.

Design of the composite service - the following steps need to be accomplished by a third party service provider:

- Declaring the interfaces for all the invoked components (Agenda, Localization, Yellow Pages, 3rd Party Call),
- Declaring the composite component - with a single 'orchestrate' operation - and defining the logic of this operation through a state machine.

Figure 9.11 shows the interface of the Agenda component which abstracts and simplifies a piece of functionality common to various online calendar tools.

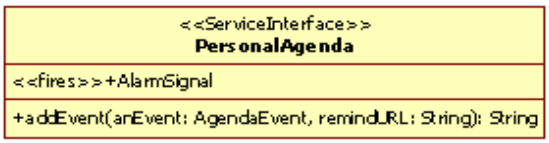


Figure 9.11 - Interface of an invoked component

Figure 9.12 shows an excerpt of the model of the service logic of the orchestration operation where three threads of execution are observed.

Note: Interface definitions are not detailed here as well as the declaration of some of the variables being used.

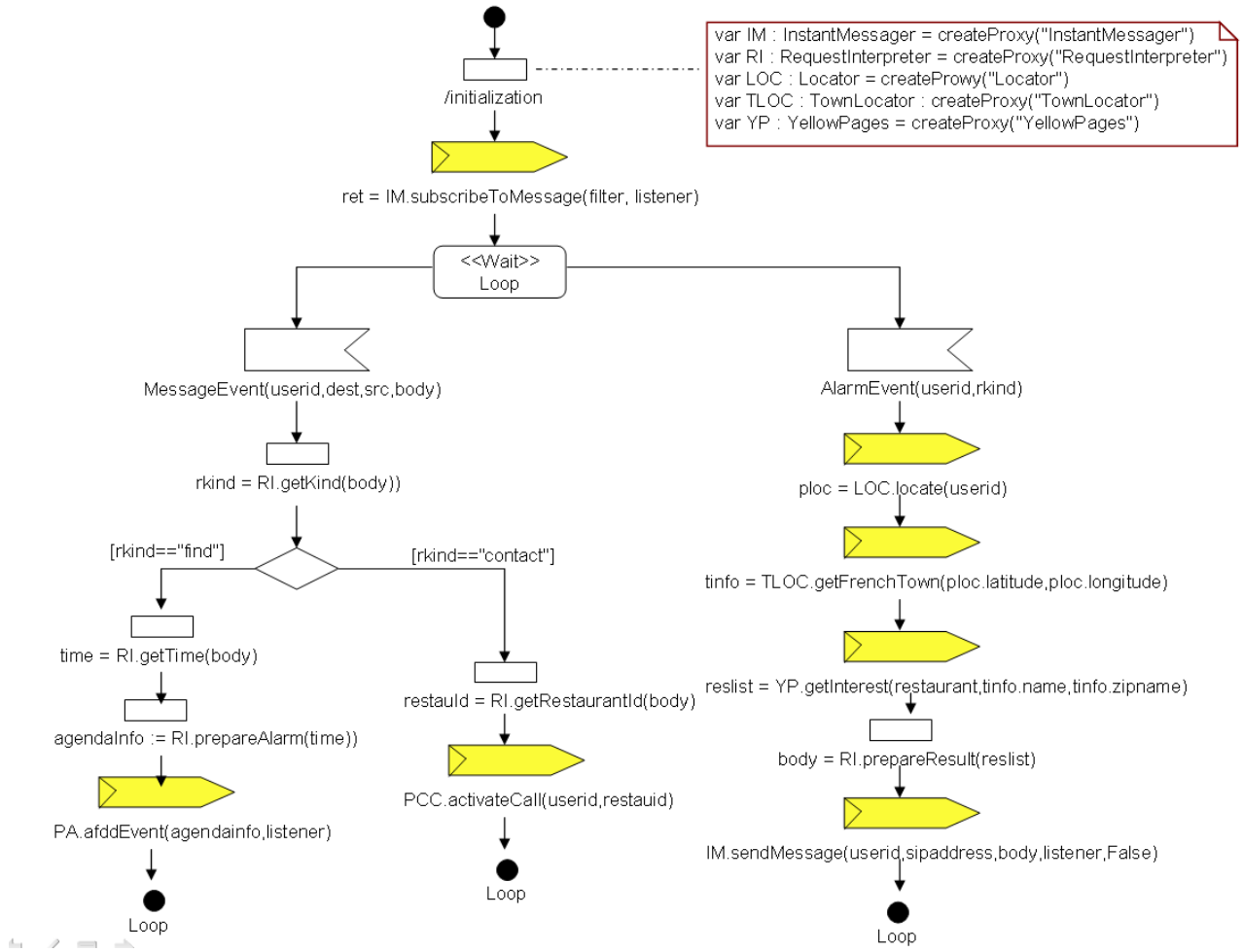


Figure 9.12 - Logic of the dinner planning composite service

On the left branch of the diagram, user's initial request is received, while on the right branch, processing of the event triggered at dinner time is performed and in the middle branch the final phone call is placed. Note that this state machine uses the UML2 transition centric view in which the actions executed during the triggering of a transition are explicitly represented as rectangles. In this diagram a specific icon is used to denote a remote service invocation, similar to an asynchronous signal sending symbol in UML.

In this diagram the two short-hand notations presented in “Additional presentation options” on page 64 have been used.

Annex A: SES/SMI Draft

(non-normative)

The SES/SMI specification from TMF is still a draft but is expected to be completed before the finalization of this specification. That's why technical elements are actually provided in a non normative annex.

More information on SES/SMI initiative can be found at:

http://www.tmforum.org/SoftwareEnabledServices/4664/home.html

A.1 Overview

The SMI interface provides a list of operations which interact with the management applications to manage deployed services and control their execution.

A.1.1 Context

In order for the service interfaces to be re-usable and manageble in different service/business context the TM Forum SES Management Solution specifies a hook to allow consistent access to the software components for OAM tasks in order to achieve consistent end-to-end management for the Service Providers.

This consistent access is achieved by incorporating the SES Management Interface (SMI) as part of the software component creation. In addition, the SMI operation is supported by Lifecycle Management Metadata (LMM) defined in the SES Reference Architecture. The LMM allows multiple stakeholders within a service ecosystem to access, create and manipulate management information per applicable business scenario. The SES design patterns enable reusability of services in different environments including Cloud by manipulating the LMM.

A.1.2 Technical Definition in SoaML

The figure below provides the interface definition. Service operations defined by this interfaces uses various data types which are also formalized in UML.

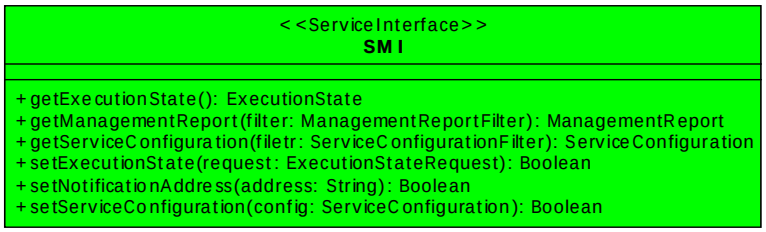


Figure A.1 - SMI Service Interface

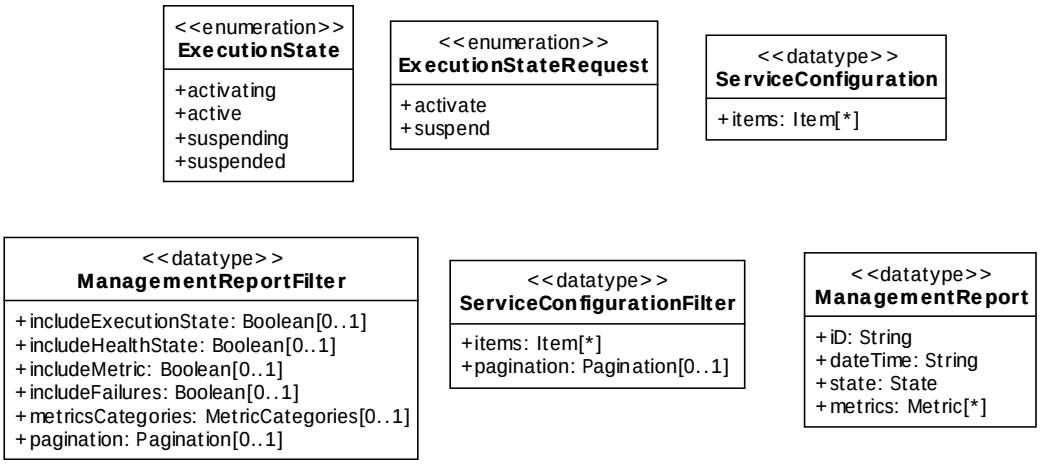


Figure A.2 - SMI Data Types (first level)

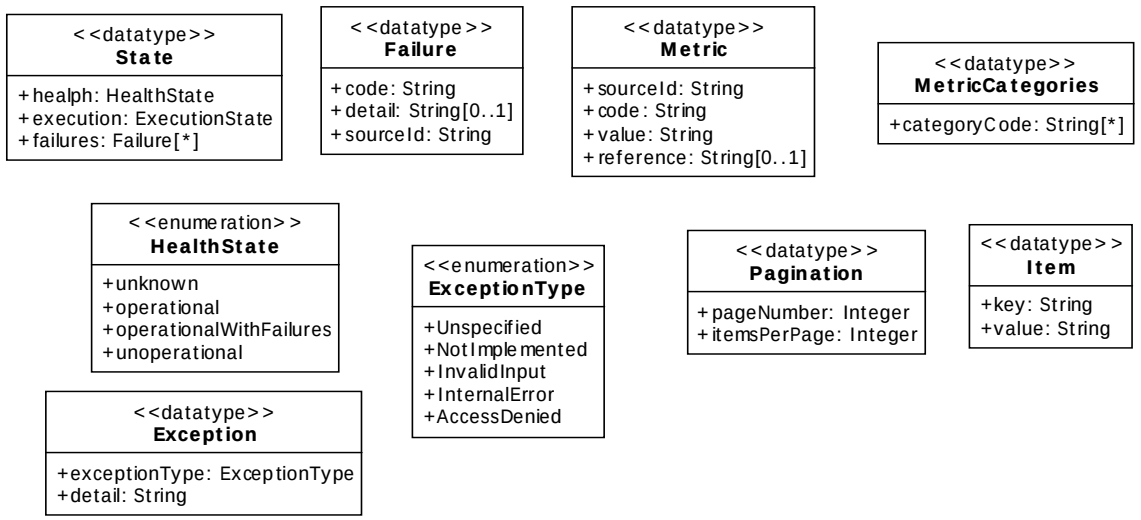


Figure A.3 - SMI Data types (second level)

A.2 Semantics of service operations

For convenience, we provide below the summary description of the operations as defined by the SMI TMF draft document. More details need to be found in the TMF document.

Retrieving the execution state

`getExecutionState(): ExecutionState`

Returns an *ExecutionState*, describing the current execution state of a service instance.

Retrieving the management report

`getManagementReport(filter: ManagementReportFilter): ManagementReport`

Returns a *ManagementReport* containing information about the service instance health, execution state, eventual failures and metrics (usage, performance for example). Optionally, it accepts a *ManagementReportFilter* input parameter that allows the service consumer to select which information will be returned and also to control the amount of Metrics returned, using a *Pagination* element in the *ManagementReportFilter.* The operation will return a complete *ManagementReport* when the parameter *ManagementReportFilter* is not received.

Retrieving the service configuration

`getServiceConfiguration(filetr: ServiceConfigurationFilter): ServiceConfiguration`

Returns a *ServiceConfiguration* containing a list of pairs Key/Value that describe the current set configuration values used by the service instance. Optionally, it accepts a *ServiceConfigurationFilter* input parameter that allows the service consumer to select which information will be returned and also to control the amount of configuration values returned, using a *Pagination* element in the *ServiceConfigurationFilter.* The operation will return a complete *ServiceConfiguration* when the parameter *ServiceConfigurationFilter* is not received.

Setting the execution state

`setExecutionState(request: ExecutionStateRequest): Boolean`

Allows a service consumer to activate or suspend service execution. It has one output parameter (a Boolean) that will return the value *True* if the change of service execution state requested by the consumer was made successfully. It has one input parameter (to activate or suspend the service instance).

Setting the notification address (register the listener)

`setNotificationAddress(address: String): Boolean`

It has one input parameter that configures the communication endpoint address the service instance must use to deliver a report containing information about their health, execution state, eventual failures and metrics. It has one output parameter (a Boolean) that will return the value *True* if the notification address configuration requested by the service consumer was made successfully.

Setting the service configuration

`setServiceConfiguration(config: ServiceConfiguration): Boolean`

Applies configuration values used by the service instance. It requires ServiceConfiguration input parameters that describe the configuration values to be applied to the service instance.