ETSI TS 129 198-1 V5.4.0 (2003-12)

Technical Specification

Universal Mobile Telecommunications System (UMTS); Open Service Access (OSA); Application Programming Interface (API); Part 1: Overview (3GPP TS 29.198-01 version 5.4.0 Release 5)

Reference
RTS/TSGN-0529198-01v540

Keywords
UMTS

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

Individual copies of the present document can be downloaded from:
http://www.etsi.org

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
http://portal.etsi.org/tb/status/status.asp

If you find errors in the present document, send your comment to:
editor@etsi.org

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by ETSI 3rd Generation Partnership Project (3GPP).

The present document may refer to technical specifications or reports using their 3GPP identities, UMTS identities or GSM identities. These should be interpreted as being references to the corresponding ETSI deliverables.

The cross reference between GSM, UMTS, 3GPP and ETSI identities can be found under http://webapp.etsi.org/key/queryform.asp .

Contents

Foreword

This Technical Specification has been produced by the 3rd Generation Partnership Project (3GPP).

The contents of the present document are subject to continuing work within the TSG and may change following formal TSG approval. Should the TSG modify the contents of the present document, it will be re-released by the TSG with an identifying change of release date and an increase in version number as follows:

Version x.y.z

where:

- x the first digit:
  - 1 presented to TSG for information;
  - 2 presented to TSG for approval;
  - 3 or greater indicates TSG approved document under change control.
- y the second digit is incremented for all changes of substance, i.e. technical enhancements, corrections, updates, etc.
- z the third digit is incremented when editorial only changes have been incorporated in the document.

Introduction

This specification has been defined jointly between ETSI TISPAN, 3GPP TSG CN WG5 and The Parlay Group (http://www.parlay.org), in co-operation with a number of JAIN™ (http://www.java.sun.com/products/jain) member companies.

The present document is part 1 of a multi-part TS covering the 3rd Generation Partnership Project: Technical Specification Group Core Network; Open Service Access (OSA); Application Programming Interface (API), as identified below. The **API specification** (3GPP TS 29.198) is structured in the following Parts:

Part 1:	**"Overview";**	
Part 2:	"Common Data Definitions";	
Part 3:	"Framework";	
Part 4:	"Call Control";	
	Sub-part 1: "Call Control Common Definitions";	(new in 3GPP Release 5)
	Sub-part 2: "Generic Call Control SCF";	(new in 3GPP Release 5)
	Sub-part 3: "Multi-Party Call Control SCF";	(new in 3GPP Release 5)
	Sub-part 4: "Multi-Media Call Control SCF";	(new in 3GPP Release 5)
	Sub-part 5: "Conference Call Control SCF";	(not part of 3GPP Release 5)
Part 5:	"User Interaction SCF";	
Part 6:	"Mobility SCF";	
Part 7:	"Terminal Capabilities SCF";	
Part 8:	"Data Session Control SCF";	
Part 9:	"Generic Messaging SCF";	(not part of 3GPP Release 5)
Part 10:	"Connectivity Manager SCF";	(not part of 3GPP Release 5)
Part 11:	"Account Management SCF";	
Part 12:	"Charging SCF".	
Part 13:	"Policy Management SCF";	(new in 3GPP Release 5)
Part 14:	"Presence and Availability Management SCF";	(new in 3GPP Release 5)

The **Mapping specification of the OSA APIs and network protocols** (3GPP TR 29.998) is also structured as above. A mapping to network protocols is however not applicable for all Parts, but the numbering of Parts is kept. Also in case a Part is not supported in a Release, the numbering of the parts is maintained.

Table: Overview of the OSA APIs & Protocol Mappings 29.198 & 29.998-family

<table>
<tr><th colspan="5">OSA API specifications 29.198-family</th><th colspan="2">OSA API Mapping - 29.998-family</th></tr>
<tr><td>29.198-01</td><td colspan="4">Overview</td><td>29.998-01</td><td>Overview</td></tr>
<tr><td>29.198-02</td><td colspan="4">Common Data Definitions</td><td>29.998-02</td><td>Not Applicable</td></tr>
<tr><td>29.198-03</td><td colspan="4">Framework</td><td>29.998-03</td><td>Not Applicable</td></tr>
<tr><td rowspan="4">Call Control (CC) SCF</td><td rowspan="4">29.198-04-1 Common CC data definitions</td><td rowspan="4">29.198-04-2 Generic CC SCF</td><td rowspan="4">29.198-04-3 Multi-Party CC SCF</td><td rowspan="4">29.198-04-4 Multi-media CC SCF</td><td>29.998-04-1</td><td>Generic Call Control – CAP mapping</td></tr>
<tr><td>29.998-04-2</td><td>Generic Call Control – INAP mapping</td></tr>
<tr><td>29.998-04-3</td><td>Generic Call Control – Megaco mapping</td></tr>
<tr><td>29.998-04-4</td><td>Multiparty Call Control – SIP mapping</td></tr>
<tr><td rowspan="4">29.198-05</td><td rowspan="4" colspan="4">User Interaction SCF</td><td>29.998-05-1</td><td>User Interaction – CAP mapping</td></tr>
<tr><td>29.998-05-2</td><td>User Interaction – INAP mapping</td></tr>
<tr><td>29.998-05-3</td><td>User Interaction – Megaco mapping</td></tr>
<tr><td>29.998-05-4</td><td>User Interaction – SMS mapping</td></tr>
<tr><td>29.198-06</td><td colspan="4">Mobility SCF</td><td>29.998-06</td><td>User Status and User Location – MAP mapping</td></tr>
<tr><td>29.198-07</td><td colspan="4">Terminal Capabilities SCF</td><td>29.998-07</td><td>Not Applicable</td></tr>
<tr><td>29.198-08</td><td colspan="4">Data Session Control SCF</td><td>29.998-08</td><td>Data Session Control – CAP mapping</td></tr>
<tr><td>29.198-09</td><td colspan="4">Generic Messaging SCF</td><td>29.998-09</td><td>Not Applicable</td></tr>
<tr><td>29.198-10</td><td colspan="4">Connectivity Manager SCF</td><td>29.998-10</td><td>Not Applicable</td></tr>
<tr><td>29.198-11</td><td colspan="4">Account Management SCF</td><td>29.998-11</td><td>Not Applicable</td></tr>
<tr><td>29.198-12</td><td colspan="4">Charging SCF</td><td>29.998-12</td><td>Not Applicable</td></tr>
<tr><td>29.198-13</td><td colspan="4">Policy Management SCF</td><td>29.998-13</td><td>Not Applicable</td></tr>
<tr><td>29.198-14</td><td colspan="4">Presence & Availability Management SCF</td><td>29.998-14</td><td>Not Applicable</td></tr>
</table>

The following table explains how the various releases of ETSI, Parlay and 3GPP OSA specifications correspond. Each ETSI and 3GPP specification carries a version number and is updated independently. The frequency of 3GPP updates may be up to every 3 months, which is greater than that of ETSI or Parlay, therefore, while there is a corresponding version of 3GPP TS 29.198 for every version of ETSI ES 201 915 or ES 202 915, there is not necessarily a corresponding version of the ETSI specification for each version of the 3GPP specification. For example, there is no ETSI/Parlay specification version which corresponds exactly to the 3GPP issue of TS 29.198 Release 4 from December 2001.

ETSI ES 201 915 / Parlay 3 / 3GPP TS 29.198 Release 4 (version 4.x.x)

ETSI OSA Specification Set	Parlay Phase	3GPP TS 29.198 version
-	-	Release 4, March 2001 Plenary
-	-	Release 4, June 2001 Plenary
ES 201 915 v.1.1.1 (complete release)	Parlay 3.0	Release 4, September 2001 Plenary
-	-	Release 4, December 2001 Plenary
ES 201 915 v.1.2.1 (complete release)	Parlay 3.1	Release 4, March 2002 Plenary
ES 201 915 v.1.3.1 (complete release)	Parlay 3.2	Release 4, June 2002 Plenary
-	-	Release 4, September 2002 Plenary
ES 201 915 v.1.4.1 (complete release)	Parlay 3.3	Release 4, March 2003 Plenary
-	-	Release 4, June 2003 Plenary
-	-	Release 4, December 2003 Plenary

ETSI ES 202 915 / Parlay 4 / 3GPP TS 29.198 Release 5 (version 5.x.x)

ETSI OSA Specification Set	Parlay Phase	3GPP TS 29.198 version
-	-	Release 5, March 2002 Plenary
ES 202 915 v.1.1.1 (complete release)	Parlay 4.0	Release 5, September 2002 Plenary
ES 202 915 v.1.2.1 (not parts 9, 13, 14)	Parlay 4.1	Release 5, March 2003 Plenary
-	-	Release 5, June 2003 Plenary
-	-	Release 5, September 2003 Plenary
-	-	Release 5, December 2003 Plenary

ETSI ES 203 915 / Parlay 5 / 3GPP TS 29.198 Release 6 (version 6.x.x)

ETSI OSA Specification Set	Parlay Phase	3GPP TS 29.198 version
-	-	Release 6, June 2003 Plenary
-	-	Release 6, December 2003 Plenary

1 Scope

The present document is the first part of the 3GPP Specification defining the Application Programming Interface (API) for Open Service Access (OSA), and provides an overview of the content and structure of the various parts of this specification, and of the relation to other standards documents .

The OSA-specifications define an architecture that enables service application developers to make use of network functionality through an open standardised interface, i.e. the OSA APIs. The concepts and the functional architecture for the OSA are contained in 3GPP TS 23.127 [3]. The requirements for OSA are contained in 3GPP TS 22.127 [2].

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication, edition number, version number, etc.) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies. In the case of a reference to a 3GPP document (including a GSM document), a non-specific reference implicitly refers to the latest version of that document *in the same Release as the present document*.

[1] 3GPP TR 21.905: "Vocabulary for 3GPP Specifications".

[2] 3GPP TS 22.127: "Service Requirement for the Open Services Access (OSA)".

[3] 3GPP TS 23.127: "Virtual Home Environment / Open Service Access (OSA)".

[4] Void.

[5] 3GPP TS 22.101: "Service Aspects; Service Principles".

[6] Void.

[7] 3GPP TS 29.002: "Mobile Application Part (MAP) specification".

[8] 3GPP TS 29.078: "Customised Applications for Mobile network Enhanced Logic (CAMEL); CAMEL Application Part (CAP) specification".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the terms and definitions given in 3GPP TS 22.101 [5] and the following apply.

Applications: Services, which are designed using Service Capability Features (SCFs).

Gateway: Synonym for Service Capability Server (SCS). From the viewpoint of applications, an SCS can be seen as a gateway to the core network.

HE-VASP: Home Environment Value Added Service Provider. This is a VASP that has an agreement with the Home Environment to provide services.

Home Environment: responsible for overall provision of services to users.

Local Service: A service, which can be exclusively provided in the current serving network by a Value Added Service Provider.

OSA Interface: Standardised Interface used by application to access service capability features.

Personal Service Environment (PSE): contains personalised information defining how subscribed services are provided and presented towards the user. The Personal Service Environment is defined in terms of one or more User Profiles.

Service Capabilities: Bearers defined by parameters, and/or mechanisms needed to realise services. These are within networks and under network control.

Service Capability Feature (SCF): Functionality offered by service capabilities that are accessible via the standardised OSA interface.

Service Capability Server (SCS): Functional Entity providing OSA interfaces towards an application.

Service: term used as an alternative for Service Capability Feature in this specification.

User Interface Profile: Contains information to present the personalised user interface within the capabilities of the terminal and serving network.

User Profile: This is a label identifying a combination of one user interface profile, and one user services profile.

User Services Profile: Contains identification of subscriber services, their status and reference to service preferences.

Value Added Service Provider: provides services other than basic telecommunications service for which additional charges may be incurred.

Virtual Home Environment: A concept for personal service environment portability across network boundaries and between terminals.

3.2 Abbreviations

For the purposes of the present document, the abbreviations given in 3GPP TR 21.905 [1] and the following apply.

API	Application Programming Interface
CAMEL	Customised Application for Mobile network Enhanced Logic
CAP	CAMEL Application Part
CSE	CAMEL Service Environment
FW	Framework
HE	Home Environment
HE-VASP	Home Environment - Value Added Service Provider
HLR	Home Location Register
INAP	Intelligent Networks Application Part
IDL	Interface Description Language
JSR	Java Specification Request
MAP	Mobile Application Part
ME	Mobile Equipment
MExE	Mobile Station (Application) Execution Environment
MS	Mobile Station
MSC	Mobile Switching Centre
OSA	Open Service Access
PLMN	Public Land Mobile Network
PSE	Personal Service Environment
RMI	Java Remote Method Invocation
SAT	SIM Application Tool-Kit
SCF	Service Capability Feature
SCP	Service Control Point
SCS	Service Capability Server
SIM	Subscriber Identity Module
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol

SOAP	Simple Object Access Protocol
SPA	Service Provider API
UE	User Equipment
USIM	Universal Subscriber Identity Module
VLR	Visited Location Register
VASP	Value Added Service Provider
VHE	Virtual Home Environment
WAP	Wireless Application Protocol
WGP	Wireless Gateway Proxy
WPP	Wireless Push Proxy
WSDL	Web Services Definition Language
XML	Extensible Markup Language

4 Open Service Access APIs

The OSA-specifications define an architecture that enables service application developers to make use of network functionality through an open standardised interface, i.e. the OSA APIs. The network functionality is describes as Service Capability Features (SCFs) or Services. The OSA Framework is a general component in support of Services (Service Capabilities) and Applications. The concepts and the functional architecture for the OSA are contained in 3GPP TS 23.127 [3]. The requirements for OSA are contained in 3GPP TS 22.127 [2].

The OSA API is split into three types of interface classes, Service and Framework (FW).

- Interface classes between the Applications and the Framework (FW), that provide applications with basic mechanisms (e.g. Authentication) that enable them to make use of the service capabilities in the network.

- Interface classes between Applications and SCFs, which are individual services that may be required by the client to enable the running of third party applications over the interface e.g. Messaging type service.

- Interface classes between the Framework (FW) and the SCFs, that provide the mechanisms necessary for a multi-vendor environment.

These interfaces represent interfaces 1, 2 and 3 in Figure 1 below. The other interfaces are not yet part of the scope of the work.

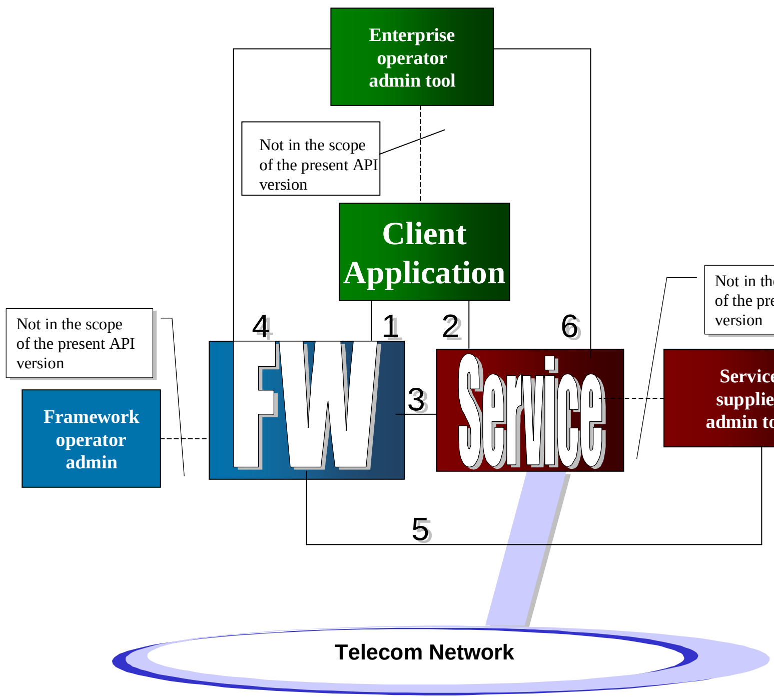


Figure 1:

Within the OSA concept a set of Service Capability Features (SCFs) has been specified. The OSA documentation is structured in parts. The first Part (the present document) contains an overview, the second Part contains common data definitions, the third Part the Framework interfaces and the following Parts contain the description of the SCFs.

NOTE: The terms 'Service' and 'Service Capability Feature' are used as alternatives for the same concept in the present document. In the OSA API itself the SCFs as identified in the 3GPP requirements and architecture are reflected as 'service', in terms like service instance lifecycle manager, service Discovery.

5 Structure of the OSA API (29.198) and Mapping (29.998) documents

The Open Service Access (OSA) Application Programming Interface (API) specifications consist of two sets of documents:

API specification (3GPP TS 29.198)
The Parts of 29.198 - apart from Part 1 (the present document) and Part 2 - define the interfaces, parameters and state models that belong to the API specification. UML (Unified Modelling Language) is used to specify the interface classes.
As such it provides a UML interface class description of the methods (API calls) supported by that interface and the relevant parameters and types. The interfaces are specified both in IDL (Interface Description Language) and in WSDL (Web Services Definition Language). Reference is made to the Java API specification of the interfaces.

Mapping specification of the OSA APIs and network protocols (3GPP TR 29.998)
The Parts of 29.998 contain a possible mapping from the APIs defined in 29.198 to various network protocols (i.e. MAP [7], CAP [8], etc.). It is an informative document, since this mapping is considered as implementation- / vendor-dependent. On the other hand this mapping will provide potential service designers with a better understanding of the relationship of the OSA API interface classes and the behaviour of the network associated to these interface classes.

The purpose of the OSA API is to shield the complexity of the network, its protocols and specific implementation from the applications. This means that applications do not have to be aware of the network nodes, a Service Capability Server interacts with, in order to provide the SCFs to the application. The specific underlying network and its protocols are transparent to the application.

The **API specification** (3GPP TS 29.198) is structured in the following Parts:

29.198-1	Part 1:	Overview
29.198-2	Part 2:	Common Data Definitions
29.198-3	Part 3:	Framework
29.198-4	Part 4:	Call Control SCF
29.198-5	Part 5:	User Interaction SCF
29.198-6	Part 6:	Mobility SCF
29.198-7	Part 7:	Terminal Capabilities SCF
29.198-8	Part 8:	Data Session Control SCF
29.198-9	Part 9:	Generic Messaging SCF (not part of 3GPP Release 5)
29.198-10	Part 10:	Connectivity Manager SCF (not part of 3GPP Release 5)
29.198-11	Part 11:	Account Management SCF
29.198-12	Part 12:	Charging SCF
29.198-13	Part 13:	Policy Management SCF
29.198-14	Part 14:	Presence & Availability Management SCF

The **Mapping specification of the OSA APIs and network protocols** (3GPP TR 29.998) is also structured as above. A mapping to network protocols is however not applicable for all Parts, but the numbering of Parts is kept.
Also in case a Part is not supported in a Release, the numbering of the parts is maintained.

Structure of the Parts of 29.198

The Parts with API specification themselves are structured as follows:

- The Sequence diagrams give the reader a practical idea of how each of the SCF is implemented.
- The Class relationships clause shows how each of the interfaces applicable to the SCF, relate to one another.
- The Interface specification clause describes in detail each of the interfaces shown within the Class diagram part.
- The State Transition Diagrams (STD) show the progression of internal processes either in the application, or Gateway.
- The Data definitions clauses show a detailed expansion of each of the data types associated with the methods within the classes. It is to be noted that some data types are used in other methods and classes and are therefore defined within the Common Data types part of this specification.

The OSA API is defined using UML and as such is technology independent. OSA can be realised in a number of ways and in addition to the UML defined OSA API, the OSA specification includes:

- A normative annex with the OSA API in IDL that specifies the CORBA distribution technology realisation
- An informative annex with the OSA API in WSDL that specifies the SOAP/HTTP distribution technology realisation
- An informative annex that references the OSA API in Java (known as JAIN™ Service Provider API) that specifies the Java local API technology realisation

6 Methodology

Following is a description of the methodology used for the establishment of API specification for OSA.

6.1 Tools and Languages

The Unified Modelling Language (UML) (http://www.omg.org/uml) is used as the means to specify class and state transition diagrams.

6.2 Packaging

A hierarchical packaging scheme is used to avoid polluting the global name space. The root is defined as:

org.csapi

6.3 Colours

For clarity, class diagrams follow a certain colour scheme. Blue for application interface packages and yellow for all the others.

6.4 Naming scheme

The following naming scheme is used for documentation.

packages

lowercase.

Using the domain-based naming (For example, org.csapi)

classes, structures and types. Start with T

TpCapitalizedWithInternalWordsAlsoCapitalized

Exception class:

TpClassNameEndsWithException and P_UPPER_CASE_WITH_UNDERSCORES_AND_START_WITH_P

Interface. Start with Ip:

IpThisIsAnInterface

constants:

P_UPPER_CASE_WITH_UNDERSCORES_AND_START_WITH_P

methods:

firstWordLowerCaseButInternalWordsCapitalized()

method's parameters

firstWordLowerCaseButInternalWordsCapitalized

collections (set, array or list types)

TpCollectionEndsWithSet

class/structure members

FirstWordAndInternalWordsCapitalized

Spaces in-between words are not allowed.

6.5 State Transition Diagram text and text symbols

The descriptions of the State Transitions in the State Transition Diagrams follow the convention:

when_this_event_is_received [guard condition is true] /do_this_action ^send_this_message

Furthermore, text underneath a line through the middle of a State indicates an exit or entry event (normally specified which one).

6.6 Exception handling and passing results

OSA methods communicate errors in the form of exceptions. OSA methods themselves always use the return parameter to pass results. If no results are to be returned a void is used instead of the return parameter. In order to support mapping to as many languages as possible, no method *out* parameters are allowed.

6.7 References

In the interface specification whenever Interface parameters are to be passed as an *in* parameter, they are done so by reference, and the "Ref" suffix is appended to their corresponding type (e.g. IpAnInterfaceRef anInterface), a reference can also be viewed as a logical indirection.

Table:

Original type	IN parameter declaration
IpInterface	parm : IN IpInterfaceRef

6.8 Strings and Collections

For character strings, the *String* data type is used without regard to the maximum length of the string.

For homogeneous collections of instances of a particular data type the following naming scheme is used: <datatype>Set

6.9 Prefixes

OSA constants and data types are defined in the global name space: *org.csapi*.

7. Introduction to OSA APIs

This section contains the general rules that were followed by the design of the OSA APIs and advice for how to use them. Note however that exceptions to these “rules” may exist and that examples are not exhaustive.

7.1 Interface Types

In the OSA specifications different types of interfaces are distinguished:

- Application side (callback) interfaces. This type of interface needs to be implemented by an application (client) and the name of such an interface is prefixed with “IpApp”.
- Interfaces of an SCF that are used by the Framework. The name of this type of server interface is prefixed with “IpSvc”.
- Application side interfaces and SCF interfaces that are shared. The name of this type of interface is prefixed with “IpClient”
- Interfaces of the Framework that are used by an SCF. The name of this type of server interface is prefixed with “IpFw”.

The name of all other interfaces of the Framework and SCFs that are used by an application, is prefixed with “Ip”.

7.2 Service Factory

For each application that uses an SCF, a separate object is created to handle all communication to the application. This object is referred to as the Service Manager. The pattern used is often referred to as the Factory Pattern. The Service Manager creates any new objects in the SCF. The Service Manager and all the objects created by it are referred to as “service instance”.

Once an application is granted access to an SCF, the Framework requests the SCF to create a new Service Manager. The reference to this Service Manager is provided to the application. From this moment onwards the application can start using the SCF.

7.3 Use of Sessions

A session is a series of interactions between two communication end points that occur during the span of a single connection. An example is all operations to set-up, control, and tear down a (multi-party) call. A session is identified by a Session ID. This ID is unique within the scope of a service instance and can be related to session numbers used in the network.

7.4 Interfaces and Sessions

Some interfaces have a one-to-one relation with a session. For every session there is a separate interface instance. In this case, this instance of an interface represents the session. All methods invoked on such an interface operate on the same session. These interfaces make no use of Session IDs.

Other interfaces can represent multiple sessions. The underlying implementation can then either create an instance per session or it can handle multiple sessions per instance (e.g., to combat extensive resource usage). When a method on such an interface is invoked it requires a Session ID to uniquely identify the session to which it applies.

7.5 Callback Interfaces

Some OSA interfaces require an application to register a callback interface. This interface resides on the client (application) side and is used by the server (service) to report events, results, and errors. An application shall register its callback interface as soon as the corresponding server side interface is created.

7.6 Setting Callbacks

Two methods are available in every service interface that can be used for setting the callback interface: setCallback() and setCallbackWithSessionID(). Interfaces that do not use sessions shall (obviously) only implement setCallback(). An invocation of setCallbackWithSessionID() on such interfaces shall result in an exception (P_TASK_REFUSED).

Interfaces that use sessions shall only implement setCallbackWithSessionID(). An invocation of setCallback() on such interfaces shall result in an exception (P_TASK_REFUSED). This regardless of whether an interface instance actually implements multiple sessions or not.

7.7 Synchronous versus Asynchronous Methods

Two types of methods exist in OSA interfaces. When a method does not require the SCS to contact other nodes in the network it is implemented as a synchronous method. When the method returns, the result (if applicable) of the operation is provided to the application. When an error occurs, an exception is thrown. Examples of synchronous methods are methods to retrieve data that is available in the SCS and methods that create an object.

In other cases, a method requires the SCS to contact other nodes in the network. There can be a delay between the moment a message is sent into the network and the moment that the result is received or an error is detected. To prevent that the application is blocked or that an application has to “guess” whether there is a problem in the SCS, these types of methods are made asynchronous.

An asynchronous method of an interface can be recognized by the fact that its name ends with “Req” (from request) and that in the corresponding callback interface two methods are included with the same name but ending with “Res” (from result) and “Err” (from error) instead. When no error has occurred, the “Res” method will be invoked when the result is available. In case an error has been detected, the “Err” method is invoked. Problems that can be detected by the SCS itself (for instance illegal parameter values) will result in exceptions being thrown when the “Req” method is called. After a “Req” method has returned, only errors shall be reported.

Because it is possible that multiple requests can be done in parallel (invoking multiple times a “Req” method without having received a result or error) a mechanism is needed to link requests with responses. Therefore, the “Req” method returns an Assignment ID and the “Res” and “Err” methods have this Assignment ID as input parameter. For session based interfaces the Session ID can be used also.

Some “Req” methods can result in multiple “Res” methods being invoked. However, the corresponding “Err” method will never be invoked more than once.

Note that methods on client side interfaces shall never raise an exception unless this is explicitly described in the specification.

Some methods switch on/off reports (for instance triggered location reports). These methods are of a different kind and do not follow the pattern that is described in this section.

A deadlock is a potential danger when using asynchronous methods, especially in single threaded systems. It can occur that client and server are waiting for each other for a task to be completed. It is considered good practice to build in mechanisms to prevent deadlock from occurring, for instance by using multiple threads or using time-outs on remote method calls.

7.8 Out Parameters

Methods used in OSA interfaces only have input parameters. Any result can only be reported by a return value. If multiple values need to be returned, a datatype is required that consists of a sequence of values. A value of this datatype is then returned by a method. This approach has been chosen because not all middleware solutions are (or may be) capable of dealing with (multiple) output parameters.

7.9 Exception Hierarchy

Exceptions are organized in an exception hierarchy. For the general exceptions and for each service type an abstract exception class is defined. Advantage for an application programmer is that (s)he does not need to catch all the specific exceptions, but may catch only the abstract exceptions.

Note however that the exception hierarchy is only available when the applicable OSA realisation supports this. Java does, but CORBA and WSDL/SOAP do not.

7.10 Common Exceptions

Exception TpCommonExceptions can be thrown by any method. It is an aggregate of a number of general problems. To prevent that each method's signature requires all these exceptions they are all included in a single exception class.

The following rules apply on when what type of general exception shall be thrown:

- P_RESOURCES_UNAVAILABLE is thrown when a physical resource in the network is not available.
- P_INVALID_STATE is thrown when a method is called that is not allowed in the state that the OSA state machines are in.
- P_TASK_CANCELLED is thrown in case of a temporary problem.
- P_TASK_NO_CALLBACK_ADDRESS_SET is thrown when no callback address has been set.
- P_METHOD_NOT_SUPPORTED is thrown when the application initiates methods that are either not according to the Service Level Agreement or not supported in the SCS.
- P_TASK_REFUSED is thrown in case of a problem that is not temporary and when none of the other common or dedicated exceptions apply.

Note that methods on application side callback interfaces shall never raise an exception unless explicitly stated in the specification.

7.11 Use of NULL

The OSA specifications contain references to the NULL value to indicate the absence of a certain parameter. An example where this is used is for specifying NULL as a callback reference.

A parameter description for parameters of any datatype can indicate that NULL is a possible value. The realisation of NULL can differ per technology. A NULL value for a sequence in CORBA means that all its members shall be NULL while in Java the whole structure could be NULL.

Note that it always shall be stated in the specification when a NULL value can be expected.

7.12 Notification Handling

Several OSA SCFs provide a mechanism for creating and receiving notifications. A notification is the reporting of an event occurring in the network or SCS. Examples of notifications are answer, busy, and on hook events.

This section describes the general mechanism of notification handling. Note that it might not apply (exactly) to every API.

There are two types of notifications. One that is created by an application and one that is controlled by the network. The first type normally is used when an ASP is responsible for service provisioning and has to create its own notifications in order to be able to serve subscribers. The second type is used when the network operator does service provisioning. The network operator creates the notifications and an application only needs to handle them.

Note that normally both mechanisms will not be used by one application. However, the OSA interfaces do not prohibit this.

Another way to distinguish notifications is by monitor mode. Notifications can be requested in either NOTIFY or INTERRUPT mode. When requested in NOTIFY mode, the notifications is reported to the application but the SCS continues processing. For notifications requested in INTERRUPT mode, processing in the SCS is suspended when the notification is reported to the application. The application has to instruct the SCS explicitly (within a certain maximum time) how to proceed the processing. Note that not all SCFs support notifications in INTERRUPT mode.

When a notification is created and when an application registers for network controlled notifications a callback interface needs to be provided. This callback interface is used for reporting the notifications. There are however a few things that are worth mentioning here:

- Each time a (set of) notifications(s) is created, a callback is specified that is used for reporting the requested notifications. This callback interface may be the same, but may also differ. The assignment ID can be used to link a notification report to the creation of registration.
- Registering a callback for network controlled notifications needs to be done only once. The callback interface that is provided may be the same as the one used for creating a notification (note again that it is however not recommended to used both mechanisms in the same application).
- The callback specified when creating or registering for events overrules the callback set with setCallback() or setCallbackWithSessionID(). This means that this one will NOT be used for reporting notifications . It will however be used for all other methods that require the callback interface.
- Only if NULL is provided as callback interface reference, the callback interface that was set using setCallback() or setCallbackWithSessionID() is used for reporting notifications.
- It is possible to recreate a (set of) notification(s) or re-register for notifications. This is only useful when providing a different callback interface reference. In this case, the last provided interface is used for reporting notifications. The earlier provided callback interface is used as “backup” interface (this can be the one provided with setCallback() or setCallbackWithSessionID() if NULL was provided initially). Notifications are reported on this interface when calls to the most recent provided callback interface fail (object providing the interface is crashed or overloaded). When re-creating or re-registering, the same assignment ID is returned.

Annex A (normative): OMG IDL

A.1 Tools and Languages

The Object Management Group's (OMG) (http://www.omg.org/) Interface Definition Language (IDL) is used as a means to programmatically define the interfaces. IDL files are either generated manually from class diagrams or by using a UML tool. In the case IDLs are manually written and/or being corrected manually, correctness has been verified using a CORBA2 (orbos/97-02-25) compliant IDL compiler, e.g. (http://java.sun.com/products/jdk/idl/index.html).

A.2 Namespace

The used namespace in CORBA IDL is org.csapi.

A.3 Object References

In CORBA IDL it is not needed to explicitly indicate a reference to an object. Where the specifications explicitly indicate a reference to an object by adding "Ref" to the object type, this addition is removed when mapped to the IDL.

Example 1:

```
struct TpMultiPartyCallIdentifier {
    IpMultiPartyCall CallReference;
    TpSessionID CallSessionID;
};
```

A.4 Mapping of Datatypes

A.4.1 Basic Datatypes

In IDL, the data type *String* is typedefed (see Note below) from the CORBA primitive *string*. This CORBA primitive is made up of a length and a variable array of byte.

NOTE: A *typedef* is a type definition declaration in IDL.

TpBoolean maps to a CORBA boolean, TpInt32 to a CORBA long, TpFloat to a CORBA float, and TpOctet to a CORBA octet.

A.4.2 Constants

All constants are mapped to a CORBA const of type TpInt32.

Example 2: const TpInt32 P_TASK_REFUSED = 14;

A.4.3 Collections

In OMG IDL, collections (Numbered Set and Numbered List) map to a sequence of the data type. A CORBA sequence is implicitly made of a length and a variable array of elements of the same type.

Example 3: typedef sequence<TpSessionID> TpSessionIDSet;

Collection types can be implemented (for example, in C++) as a structure containing an integer for the *number* part, and an array for the *data* part.

Example 4: The TpAddressSet data type may be defined in C++ as:

```
typedef struct {
    short       number;
    TpAddress   address [];
} TpAddressSet;
```

The array "address" is allocated dynamically with the exact number of required TpAddress elements based on "number".

A.4.4 Sequences

In OMG IDL sequences map to a CORBA Struct.

Example 5:
```
struct TpAddress {
    TpAddressPlan Plan;
    TpString AddrString;
    TpString Name;
    TpAddressPresentation Presentation;
    TpAddressScreening Screening;
    TpString SubAddressString;
};
```

A.4.5 Enumerations

In OMG IDL enumerations map to a CORBA enum.

Example 6:
```
enum TpAddressScreening {
    P_ADDRESS_SCREENING_UNDEFINED ,
    P_ADDRESS_SCREENING_USER_VERIFIED_PASSED,
    P_ADDRESS_SCREENING_USER_NOT_VERIFIED,
    P_ADDRESS_SCREENING_USER_VERIFIED_FAILED ,
    P_ADDRESS_SCREENING_NETWORK
};
```

A.4.6 Choices

A choice maps to a CORBA union. For entries that do not have a corresponding type (defined as NULL in the specification) no union entry is generated. These entries are grouped in the default clause where NULL is replaced by short and the entry name (Undefined) by the name Dummy. When there are no NULL entries, the default clause is not generated.

Example 7:
```
union TpCallAdditionalErrorInfo switch (TpCallErrorType) {
    case P_CALL_ERROR_INVALID_ADDRESS: TpAddressError CallErrorInvalidAddress;
    default: short Dummy;
};
```

Example 8:
```
union TpCallChargeOrder switch(TpCallChargeOrderCategory) {
    case P_CALL_CHARGE_TRANSPARENT: TpOctetSet TransparentCharge;
    case P_CALL_CHARGE_PREDEFINED_SET: TpInt32 ChargePlan;
};
```

A.5 Use of NULL

CORBA allows the value NULL to be used for object references only. When the specification mentions NULL as possible value of a struct, it means that each object reference in the struct shall be set to NULL. NULL does not apply to other datatypes then object references.

A.6 Exceptions

The TpCommonExceptions is mapped to a CORBA exception containing a data item of type TpInt32 to indicate the type of general exception and extra information of type TpString.

Example 9: exception TpCommonExceptions {
TpInt32 ExceptionType;
TpString ExtraInformation;
};

All other exceptions are also mapped to CORBA exceptions but containing a data item of type TpString to indicate additional information.

Example 10: exception P_INVALID_ASSIGNMENT_ID {
TpString ExtraInformation;
};

A.7 Naming space across CORBA modules

The following shows the naming space used in this specification.

```
module org {
module csapi {
/* The fully qualified name of the following constant is org::csapi::P_THIS_IS_AN_OSA_GLOBAL_CONST
*/
const long P_THIS_IS_AN_OSA_GLOBAL_CONST= 1999;
// Add other OSA global constants and types here
module fw {
/* no scoping required to access  P_THIS_IS_AN_OSA_GLOBAL_CONST */
const long P_FW_CONST= P_THIS_IS_AN_OSA_GLOBAL_CONST;
};
module mm {
// scoping required to access P_FW_CONST
const long P_M_CONST= fw::P_FW_CONST;
};
};
};
```

Annex B (informative): W3C WSDL

B.1 Tools and Languages

The W3C (http://www.w3c.org) WSDL (Web Services Definition Language) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. WSDL files are generated from the UML model using scripts. The generated WSDL files are verified using WSDL compilers.. The WSDL is based on W3C WSDL 1.1

B.2 Proposed Namespaces for the OSA WSDL

Namespaces are an important part of an XML Schema. They are used to qualify the source of a particular XML element.

There are several XML/SOAP/WSDL related Namespaces which are used within each of the WSDL documents. The Namespace Prefix and the Namespace are noted below.

xmlns:wsdl = 'http://schemas.xmlsoap.org/wsdl/'

xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'

xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'

xmlns:xsd:='http://www.w3c.org/2001/XMLSchema'

There are also OSA specific namespaces which are used within the OSA WSDL documents. The OSA related namespaces present within each WSDL document depends on the WSDL document and which WSDL documents it imports. The guidelines used to derive these namespaces are:

- The root namespace for the OSA WSDL and XML schemas is http://www.csapi.org/
- There is one document generated for each component (Module) within the Analysis UML model. The document will have the name of the UML component with the extension '.wsdl' For each wsdl document generated the following additional namespaces will be included:
    - xmlns:<component name>='http://www.csapi.org/<component name>/wsdl'
    - xmlns:<component name>xsd='http://www.csapi.org/<component name>/schema'
    - For each OSA wsdl document which is referenced by an import statement within the current wsdl document then the following additional namespaces will be included.
        - xmlns:<imported component name>='http://www.csapi.org/<imported component name>/wsdl'
        - xmlns:<imported component name>xsd='http://www.csapi.org/<imported component name>/schema'
- Attributes which require a QName value shall use the appropriate Namespace Prefix (as defined in the definitions element of the wsdl file) to qualify the element being referenced.

The namespaces are defined within the 'definitions' element of a wsdl document. For example, the definitions element of the am.wsdl document would look like:

```
<definitions
   name='am'
   targetNamespace='http://www.csapi.org/am/wsdl'
   xmlns='http://schemas.xmlsoap.org/wsdl/'
   xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
```

```
   xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
   xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema'
   xmlns:am='http://www.csapi.org/am/wsdl'
   xmlns:amxsd='http://www.csapi.org/am/schema'
   xmlns:osa='http://www.csapi.org/osa/wsdl'
   xmlns:osaxsd='http://www.csapi.org/osa/schema'>

<import namespace='http://www.csapi.org/osa/wsdl'
        location='osa.wsdl' />
```

B.3 Object References

Object references are used to identify an particular remote object instance. Object references are used in two ways:

1. Passed as a parameter within a method to a remote object or passed as an attribute of a structured type parameter within a method to the remote object.
2. Included within a message to identify the object for which the message is intended.

Within the context of SOAP, an object reference can be represented as a URL appended with a String. The String suffix identifies the particular remote object instance in the context of the URL.

An object reference will be represented by the new type ObjectRef. The ObjectRef type is defined within osa.wsdl as:

```
<xsd:simpleType name='Objectref'>
   <xsd:restriction base='xsd:string' />
</xsd:simpleType>
```

When an object reference is passed as a parameter, the parameter type is defined as a reference to an interface. When an object reference is an attribute of a structured type, that attribute is defined as a reference to an interface. Each interface will have a corresponding reference type associated with it. The interface reference will be defined as:

```
<xsd:simpleType name='InterfaceNameRef'>
   <xsd:restriction base='osaxsd:ObjectRef' />
</xsd:simpleType>
```

where ***InterfaceName*** is the name of the particular interface.

When an object reference is used to identify the intended recipient of a message, then the object reference is included in the SOAP Header element as an ObjectRefHeader. The ObjectRefHeader is defined in the osa.wsdl document as follows:

```
<message name='ObjectRefHeader'>
      <part name='header' element='osaxsd:ObjectRef' />
</message>
```

Within each method, the ObjectRefHeader is bound to the message within the wsdl soap:header element of the input message of the binding element. For example:

```
<binding name='IpAccountManagerBinding' type='am:IpAccountManager'>
      <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http' />
      <operation name='createNotification'>
         <soap:operation
soapAction='http://www.csapi.org/am/IpAccountManager#createNotification' />
         <input>
            <soap:body
                  encodingStyle='http://schemas/xmlsoap.org/soap/encoding/'
                  namespace = 'http://www.csapi.org/am.wsdl'
                  use='encoded' />
            <soap:header
                  message='osaxsd:ObjectRefHeader' part='header' />
             </input>
```

B.4 Mapping UML Data Types to XML Schema

B.4.1 Data Types

B.4.1.1 <<Constant>>

The UML Constant data type contains the following attributes:

- Name
- Constant Value

This type would then map to the following XML Schema construct:

This mapping assumes that all constants are of type TpInt32

```
<xsd:simpletype name="Name">
   <xsd:restriction base="osaxsd:TpInt32">
     <xsd:minInclusive value="Constant Value" />
     <xsd:maxInclusive value="Constant Value" />
   </xsd:restriction>
</xs:simpleType>
```

B.4.1.2 <<NameValuePair>>

The UML NameValuePair data type contains the following attributes:

- Name
- Attributes
  - Name

This type would then map to the following XML Schema construct:

```
<xsd:simpleType base="xsd:string" name="Name">
   <xsd:restriction base="xsd:String">
      <xsd:enumeration value="Attribute-Name" />
      <xsd:enumeration value="Attribute-Name" />
      …
      <xsd:enumeration value="Attribute-Name" />
   </xsd:restriction>
</xsd:simpleType>
```

B.4.1.3 <<SequenceOfDataElements>>

The UML SequenceOfDataElements data type contains the following attributes:

- Name
- Roles
  - Name
  - Type

This type would then map to the following XML Schema construct:

```
<xsd:complexType name="Name"
   <xsd:sequence>
     <xsd:element
       Name="Role-Name"
       type="Role-Type" />
     <xsd:element
       Name="Role-Name"
       type="Role-Type" />
       …
     <xsd:element
       Name="Role-Name"
       type="Role-Type" />
   </xsd:sequence>
</xsd:complexType>
```

B.4.1.4 <<TypeDef>>

The UML TypeDef data type contains the following attributes:

- Name
- ImplementationType

If the Implementation type is a technology specific type, then the following mappings have been made:

TpBoolean – xsd:boolean

TpInt32 – xsd:float

TpFloat – xsd:float

TpLongString – xsd:string

TpString – xsd:string

TpOctet – xsd:hexBinary

This type would then map to the following XML Schema construct:

```
<complexType name="Name" base="ImplementationType" />
```

B.4.1.5 <<NumberedSetOfDataElements>>

The UML NumberedSetOfDataElements data type for sequences types contains the following attributes:

- Name
- ImplementationType

This type would then map to the following XML Schema construct:

```
<xsd:complexType name="Name">
   <xsd:sequence>
      <xsd:element
        name="Name"
        type="ImplementationType"
```

```
        minOccurs="0"
        maxOccurs="unbounded" />
   </xsd:sequence>
</xsd:complexType>
```

B.4.1.6 <<TaggedChoiceOfDataElements>>

The UML TaggedChoiceOfDataElements data type contains the following attributes:

- Name
- SwitchType
- Roles
  - Name
  - Type

This type would then map to the following XML Schema construct:

```
<xsd:complexType name="Name">
   <xsd:sequence>
     <xsd:element name="SwitchName" type="SwitchType" />
     <xsd:choice>
       <xsd:element name="Role-Name" type="Role-Type" />
       <xsd:element name="Role-Name" type="Role-Type" />
       …
       <xsd:element name="Role-Name" type="Role-Type" />
     </xsd:choice>
   </xsd:sequence>
</complexType>
```

B.5 Mapping of UML SCF to WSDL

B.5.1 Mapping of Operations to WSDL *message* element

A UML Operation contains the following attributes:

- Interface
- Name
- Module Name
- Return Type
- Parameter
  - Name
  - Type

This type would then map to the following XML Schema construct:

```
<message name="Interface_Name">
   <part
```

```
     name="Parameter-Name"
     type="Parameter-Type"/>
   …
   <part
     name="Parameter-Name"
     type="Parameter-Type"/>
</message>

<message name="Interface_NameResponse">
   <part name="return" type="ReturnType"/>
</message>
```

Note: If the ReturnType is void, then no 'part' element would be included in the Response message.
(i.e. `<message name="Interface_NameResponse" />`).

B.5.2 Mapping of Exception to WSDL *message* element

A UML Exception has the following attributes:

- Name

All exceptions (except for CommonException), contain a parameter called ExtraInformation which is of type TpString.

This type would then map to the following XML Schema Construct:

```
<message name="Name">
   <part
     name="ExtraInformation"
     type="osaxsd:TpString"/>
</message>
```

B.5.3 Mapping of CommonExceptions to WSDL message element

The UML CommonExceptions type has the following attributes:

- Name ("CommonExceptions")

The UML CommonException contains two parameters; ExceptionType which is of type osaxsd:TpInt32 and ExtraInformation which is of type osaxsd:TpString.

This type would then map to the following XML Schema Construct:

```
<message name="CommonExceptions">
   <part
     name="ExceptionType"
     type="osaxsd:TpInt32" />
   <part
     name="ExtraInformation"
     type="osaxsd:TpString" />
</message>
```

B.5.4 Mapping of Interface Class to WSDL *portType* and *binding* elements

A UML Interface Class contains the following attributes:

- Name
- Associated module (i.e. component)
- Operations
    - Name
        - Parameters
            - Name
        - Exceptions
            - Name

This type would then map to the following WSDL portType element:

```
<portType name="Name">
   <operation
     name="Operation-Name"
     <input message="Operation-Name"/>
     <output message="Operation-NameResponse"/>
     <fault message="Operation–Exception– Name" />
     …
     <fault message="Operation–Exception–Name" />
   </operation>
   …
   <operation
     name="Operation-Name"
     <input message="Operation-Name"/>
     <output message="Operation-NameResponse"/>
     <fault name="Operation-Exception-Name" message="Operation–Exception–Name"
/>
     …
     <fault message="Operation–Exception–Name" />
   </operation>
</portType>
```

This type would also then map into the following WSDL binding element:

```
<binding
   name="Interface-NameBinding"
   type="Interface-Name">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

   <operation name="Operation-Name">
     <soap:operation soapAction="http://www.csapi.org/am/Name#Operation-Name"/>
     <input>
       <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://www.csapi.org/Module-Name/wsdl"
         use="encoded"/>
           <soap:header message="osaxsd:ObjRefHeader" part="header" />
     </input>
     <output>
       <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace=" http://www.csapi.org/Module-Name/wsdl "
         use="encoded"/>
     </output>
```

```
      <fault>
            <soap:fault name="Operation-Exception-Name"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://www.csapi.org/Module-Name/wsdl"
         use="encoded"/>
      </fault>

      … additional fault elements

   </operation>

   … additional operation elements

</binding>
```

B.5.5 Mapping of UML SCF to WSDL *service* element

A UML Module contains the following attributes:

- Name
- Interfaces
    - Name

This type would then map to the following WSDL service element:

```
<service name="Name">

   <port binding="Interface-NameBinding" name="Interface-Name">
     <soap:address location="http://{Service Address}"/>
   </port>

   … additional port elements
</service>

</definitions>
```

Annex C (informative): Java Realisation API

C.1 Java Realisation Overview

The Parlay/OSA UML specifications are defined in a technology neutral manner. This annex aims to deliver for Java, a developer API, provided as a realisation, supporting a Java API that represents the UML specifications.

C.1.1 J2SE API

The J2SE API supports a J2SE development environment that

- provides an abstraction of the Parlay/OSA APIs that provides a local API for J2SE developers
- supports a listener based API for SCFs and a callback API for the Framework
- uses local object references as correlation mechanisms as Java developers are familiar with object correlation
- is a local API without visibility to the underlying transport

C.1.2 J2EE API

The J2EE API supports a development environment which allows the creation of J2EE and Java RMI interfaces for both the server and client, ensuring consistent interfaces for interoperability. These interfaces may be used for Java RMI on either JRMP or IIOP (RMI/IIOP), allowing use in J2EE environments. The interfaces may also be used as a thin layer on other transports, similar to other Java technologies that provide a RMI programming interface.

The J2EE API is a suitable base for Java across Java platforms, allowing creation of implementations that:

- may be a thin layer on transport protocols
- may support J2EE remote interfaces
- may support J2EE local interfaces

The Java files created with the realisation will be made available with the Parlay/OSA specifications.

The remaining sections of this annex deal with the following areas:

- section C.2 covers the tools and languages used to produce and define the Java Realisation
- section C.3 covers the mappings that are common across both Java Realisation APIs
- section C.4 covers the mappings specific to the J2SE API
- section C.5 covers the mappings specific to the J2EE API

C 1.3 Javadoc

The Javadoc that accompanies the J2SE realisation of the Parlay/OSA API specification is provided as archive 2919801J2SE.ZIP that accompanies the present document.

The Javadoc that accompanies the J2EE realisation of the Parlay/OSA API specification is provided as archive 2919801J2EE.ZIP that accompanies the present document.

C.2 Tools and languages

The Java language is used as a means to programmatically define the interfaces. Java source files are generated automatically from UML. The Java source files are created in accordance with the mappings defined within this annex.

The generated Java source files are verified syntactically using Java compilers such as javac. The Java API comprises

- J2SE API designed to be compatible with the Java 2 SDK, Standard Edition, version 1.3 (http://java.sun.com/j2se/1.3/) or later and a
- J2EE API compatible with the Java 2 Enterprise Edition (http://java.sun.com/j2ee/).

The J2SE API, developed in accordance to the conventions defined in section C.3 and C.4 will enable:

- portable Java applications, as far as the Java API is concerned
- independence of distribution mechanism technology (e.g. CORBA,SOAP,RMI)

C.3 Generic mappings (Elements common to J2SE and J2EE)

NOTE: All Java code examples given in this section are taken from the J2SE Java Realisation API. See the appropriate Java files for examples for J2EE classes.

C.3.1 Namespace

The UML namespace org.csapi is represented by the Java package org.csapi.jr.

Packages under the org.csapi.jr package will contain "se" packages for J2SE specific Java artefacts and "ee" and “eelocal” packages for J2EE specific Java artefacts.

For example, the User Location Camel Service package structure would appear as follows:

org.csapi.jr.se.mm.ulc containing J2SE API Java artefacts

org.csapi.jr.eelocal.mm.ulc containing J2EE local API Java artefacts

org.csapi.jr.ee.mm.ulc containing the J2EE remote/RMI API Java artefacts

C.3.2 Package Naming Conventions

UML packages will be represented by Java packages. The sub-namespaces below the root namespaces described above will follow the naming used for the UML namespaces.

C.3.3 Object References

In Java there is no need to explicitly indicate a reference to an object as in Java objects are passed by value and not by reference. Where the specifications explicitly indicate a reference to an object by adding “Ref” to the object type, this addition is removed in the Java realisation.

Example 1:

UML	Java Realisation
IpUserLocationCamelRef	UserLocationCamel
IpCallRef	Call

C.3.4 Element Naming

The UML element names that begin with an uppercase will follow the Java naming conventions of with a leading lower case letter and mixed case names. The UML elements are equivalent to Java field names.

Example 2:

UML	Java Realisation
AddressPlan	addressPlan

C.3.5 Element Naming Collisions

If an element name collides with a Java keyword, the element name will be prefixed with an underscore.

Example 3:

UML	Java Realisation
Final	_final

C.3.6 Data Type Definitions

C.3.6.1 Basic Data Types

Java does not support type definitions (typedefs); therefore types are unwound to their basic data types e.g.:

Example 4:

UML	Java Realisation
TpCallAlertingMechanism	int
TpAccessType	java.lang.String

The following mappings apply to the basic data types:

UML	Java Realisation
TpBoolean	boolean
TpInt32	int
TpInt64	long
TpFloat	float
TpOctet	byte
TpString	java.lang.String
TpLongString	java.lang.String
TpAny	java.lang.Object

C.3.6.2 Constants

Constants are associated with a type definition or as a standalone entity. In both cases, the constant itself will be defined as a '`public final static`' field using its name and value.

When defined associated with a type definition, an interface using the name of the type definition will be defined enclosing all constants associated with the type definition.

Standalone constants within a package are defined within a Java interface with the name 'Constants' within that package.

Example 5:

```
package org.csapi.jr.se;
public interface Constants {
```

```
        public static final int METHOD_NOT_SUPPORTED = 22;
        public static final int NO_CALLBACK_ADDRESS_SET = 17;
        public static final int RESOURCES_UNAVAILABLE = 13;
        public static final int TASK_CANCELLED = 15;
        public static final int TASK_REFUSED = 14;
        public static final int INVALID_STATE = 744;
}
```

Example 6:

```
package org.csapi.jr.se.cc;
public interface CallSuperviseReport {
        public static final int CALL_SUPERVISE_TIMEOUT = 1;
        public static final int CALL_SUPERVISE_CALL_ENDED = 2;
        public static final int CALL_SUPERVISE_TONE_APPLIED = 4;
}
```

C.3.6.3 NumberedSetsOfDataElements (Collections)

In Java, Numbered Set and Numbered List are realised as an array of the data type.

Example 7:

UML	Java Realisation
TpAddressSet	Address[]

C.3.6.4 SequenceOfDataElements (Structures)

Struct data types are represented in Java as public final classes that implement java.io.Serializable, and have:

- each data element made available as a private variable in the class
- a default constructor and a constructor for all values are provided
- accessor and mutator methods are given for each variable
- the first letter of each sequence element name is changed to lower case
- an equals method is provided determining the equality of objects by their content
- a hashCode method is provided supporting the rules for hashCode relative to equals

Example 8:

```
package org.csapi.jr.se;
public final class Address implements java.io.Serializable {
        private AddressPlan plan;
        private String addrString = “”;
        private String name = “”;
        private AddressPresentation presentation;
        private AddressScreening screening;
        private String subAddressString = “”;

        public Address () {
        }

        public Address (AddressPlan plan, String addrString,
                String name, AddressPresentation presentation,
                AddressScreening screening, String subAddressString) {
            this.plan = plan;
            this.addrString = addrString;
            this.name = name;
            this.presentation = presentation;
            this.screening = screening;
            this.subAddressString = subAddressString;
        }

        public TpAddressPlan getPlan () {
```

```
            return (plan);
        }

        public void setPlan (TpAddressPlan plan) {
            this.plan = plan;
        }

        public String getAddrString () {
            return (addrString);
        }

        public void setAddrString (String addrString) {
            this.addrString = addrString;
        }

        … other get and set methods …

        public boolean equals (Object object) {
        // equality logic
        }

        public int hashcode () {
        // hash code calculation
        }
}
```

C.3.6.5 NameValuePair (Enumerations)

NameValuePair data types are represented in Java as public final classes that implement java.io.Serializable, and have:

- two static final data members per name-value pair
- a value returning method, named getValue()
- a name returning method, named getValueText()
- an integer conversion method, named getObject()
- a private constructor
- hashCode and equals implementations

No default constructor is provided. One of the data members per name-value pair has the same name as the name-value pair name. The other has an underscore "_" prepended and is intended for use in switch statements. Values are assigned sequentially, starting with 0.

The getObject() method returns the name-value pair class with the specified value if the specified value corresponds to an element of the name-value pair data type. If the specified value is out of range, an InvalidEnumValueException exception is raised

Example 9:

```
package org.csapi.jr.se;
public final class AddressScreening implements java.io.Serializable {
        private int _value;
        private static int _size = 5;
        private static AddressScreening[] _array = new AddressScreening[_size];


        public static final int _ADDRESS_SCREENING_UNDEFINED = 0;
        public static final AddressScreening ADDRESS_SCREENING_UNDEFINED = new
AddressScreening(_ADDRESS_SCREENING_UNDEFINED);

        public static final int _ADDRESS_SCREENING_USER_VERIFIED_PASSED = 1;
        public static final AddressScreening ADDRESS_SCREENING_USER_VERIFIED_PASSED = new
AddressScreening(_ADDRESS_SCREENING_USER_VERIFIED_PASSED);

        public static final int _ADDRESS_SCREENING_USER_NOT_VERIFIED = 2;
        public static final AddressScreening ADDRESS_SCREENING_USER_NOT_VERIFIED = new
AddressScreening(_ADDRESS_SCREENING_USER_NOT_VERIFIED);
```

```
        public static final int _ADDRESS_SCREENING_USER_VERIFIED_FAILED = 3;
        public static final AddressScreening ADDRESS_SCREENING_USER_VERIFIED_FAILED = new
AddressScreening(_ADDRESS_SCREENING_USER_VERIFIED_FAILED);

        public static final int _ADDRESS_SCREENING_NETWORK =  4;
        public static final AddressScreening ADDRESS_SCREENING_NETWORK = new
AddressScreening(_ADDRESS_SCREENING_NETWORK);

        private AddressScreening(int value) {
            this._value = value;
            this._array[this._value] = this;
        }


        public int getValue() {
            return _value;
        }

        public String getValueText() {
            switch (_value) {
            case _ADDRESS_SCREENING_UNDEFINED:
                return "ADDRESS_SCREENING_UNDEFINED";
            case _ADDRESS_SCREENING_USER_VERIFIED_PASSED:
                return "ADDRESS_SCREENING_USER_VERIFIED_PASSED";
            case _ADDRESS_SCREENING_USER_NOT_VERIFIED:
                return "ADDRESS_SCREENING_USER_NOT_VERIFIED";
            case _ADDRESS_SCREENING_USER_VERIFIED_FAILED:
                return "ADDRESS_SCREENING_USER_VERIFIED_FAILED";
            case _ADDRESS_SCREENING_NETWORK:
                return "ADDRESS_SCREENING_NETWORK";
            default:
                return "ERROR";
            }
        }

        public static AddressScreening getObject(int value) throws
org.csapi.jr.se.InvalidEnumValueException {
            if(value >= 0 && value < _size) {
                return _array[value];
            } else {
                throw new org.csapi.jr.se.InvalidEnumValueException();
            }
        }

        public boolean equals(Object o) {
        //equality logic
        }

        public int hashCode() {
        //hash code calculation
        }

}
```

C.3.6.6 TaggedChoiceOfDataElements (Unions)

Union data types are represented in Java as public final classes that implement java.io.Serializable, and have:

- a default constructor
- a discriminator field
- a discriminator accessor method, named getDiscriminator()
- an accessor and modifier method for each data element, the names of which are derived from choice element name
- hashCode and equals implementations

Conflicting names should be resolved by prefixing the field name with an underscore for getDiscriminator if there is a name clash with the mapped data type name or any of the data element names.

Where choice element type and choice element name are "NULL" and "Undefined", respectively, a Java Object set as null replaces the NULL. If multiple NULL/Undefined combinations occur in the tagged choice of data elements, the method, setUndefined, will receive the discriminator as a parameter and set _object to null.

Accessor methods shall raise an InvalidUnionAccessorException exception if the expected data element has not been set.

Example 10:

```
package org.csapi.jr.se;
public final class AoCOrder implements java.io.Serializable {
        private CallAoCOrderCategory _discriminator = null;
        private java.lang.Object _object;

        public AoCOrder() {
        }

        public CallAoCOrderCategory getDiscriminator() throws
org.csapi.jr.se.InvalidUnionAccessorException {
            if(_discriminator == null) {
                throw new org.csapi.jr.se.InvalidUnionAccessorException();
            }
            return _discriminator;
        }

        public org.csapi.jr.se.ChargeAdviceInfo getChargeAdviceInfo() throws
org.csapi.jr.se.InvalidUnionAccessorException {
            if (_discriminator != CallAoCOrderCategory.CHARGE_ADVICE_INFO) {
                throw new org.csapi.jr.se.InvalidUnionAccessorException();
            }
            return ((org.csapi.jr.se.ChargeAdviceInfo) _object);
        }

        public void setChargeAdviceInfo(org.csapi.jr.se.ChargeAdviceInfo value) {
            _discriminator = CallAoCOrderCategory.CHARGE_ADVICE_INFO;
            _object = value;
        }

        public org.csapi.jr.se.ChargePerTime getChargePerTime() throws
org.csapi.jr.se.InvalidUnionAccessorException {
            if (_discriminator != CallAoCOrderCategory.CHARGE_PER_TIME) {
                throw new org.csapi.jr.se.InvalidUnionAccessorException();
            }
            return ((org.csapi.jr.se.ChargePerTime) _object);
        }

        public void setChargePerTime(org.csapi.jr.se.ChargePerTime value) {
            _discriminator = CallAoCOrderCategory.CHARGE_PER_TIME;
            _object = value;
        }

        public java.lang.String getNetworkCharge() throws
org.csapi.jr.se.InvalidUnionAccessorException {
            if (_discriminator != CallAoCOrderCategory.CHARGE_NETWORK) {
                throw new org.csapi.jr.se.InvalidUnionAccessorException();
            }
            return ((java.lang.String) _object);
        }

        public void setNetworkCharge(java.lang.String value) {
            _discriminator = CallAoCOrderCategory.CHARGE_NETWORK;
            _object = value;
        }

        public void setUndefined(CallAocOrderCategory discriminator) {
            __discriminator = discriminator;
            __object = null;
        }

        public boolean equals(Object o) {
        //equality logic
        }

        public int hashCode() {
        //hash code calculation
        }
```

```
}
```

C.3.6.7 Exceptions

An exception maps to a constructed exception, providing appropriate constructors and accessor methods for the data contained within the exception. Each exception is defined as a public class extending java.lang.Exception, and containing a private field for each information element contained within the exception.

A default constructor is provided, along with a constructor containing only an embedded exception, a constructor containing a list of the fields in the exception and a constructor that contains the fields plus an embedded exception.

An accessor method is provided for each field, and for the embedded exception.

The following Java Realisations apply to mapping of exceptions:

- PlatformException
- P_XXX_XXX Exceptions
- TpCommonExceptions
- TpCommonExceptions' associated exceptions
- Additional abstract exceptions
- InvalidUnionAccessorException
- InvalidEnumValueException

C.3.6.7.1 PlatformException

PlatformException exception handles local platform and communication problem exceptions.

Example 11:

```
package org.csapi.jr.se;
public class PlatformException extends java.lang.RuntimeException {
        private Throwable _cause = null;

        public PlatformException () {
            super();
        }

        public PlatformException (String message) {
            super(message);
        }

        public PlatformException (String message, Throwable cause) {
            super(message);
            _cause = cause;
        }

        public PlatformException (Throwable cause) {
            _cause = cause;
        }

        public Throwable getCause() {
            return _cause;
        }
}
```

C.3.6.7.2 P_XXX_XXX Exceptions

P_XXX_XXX exceptions follow the XxxXxxException naming pattern, and inherit from java.lang.Exception.

Example 12:

```
package org.csapi.jr.se;
public class InvalidInterfaceTypeException extends java.lang.Exception {
        private Throwable _cause = null;

        public InvalidInterfaceTypeException() {
            super();
        }

        public InvalidInterfaceTypeException(String message) {
            super(message);
        }

        public InvalidInterfaceTypeException(String message,Throwable cause) {
            super(message);
            _cause = cause;
        }

        public InvalidInterfaceTypeException(Throwable cause) {
            _cause = cause;
        }

        public Throwable getCause() {
            return _cause;
        }
}
```

C.3.6.7.3 TpCommonExceptions

The name for TpCommonExceptions exception is made singular, i.e. CommonException, and inherits from java.lang.Exception.

Example 13:

```
package org.csapi.jr.se;
public class CommonException extends java.lang.Exception {
        private Throwable _cause = null;
        private int _exceptionType;
        private String _extraInformation;


        public CommonException () {
            super();
        }

        public CommonException (String message) {
            super(message);
        }
        public CommonException (String message, Throwable cause) {
            super(message);
            _cause = cause;
        }

        public CommonException (Throwable cause) {
            _cause = cause;
        }


        public Throwable getCause() {
            return _cause;
        }

        public int getExceptionType() {
            return _exceptionType;
        }

        public void setExceptionType(int exceptionType) {
            _exceptionType = exceptionType;
        }

        public String getExtraInformation() {
            return _extraInformation;
        }
```

```
        public void setExtraInformation(String extraInformation) {
            _extraInformation = extraInformation;
        }

}
```

C.3.6.7.4 TpCommonException's associated exceptions

P_XXX_XXX exception types (constants) associated with TpCommonExceptions follow the XxxXxxException naming pattern and inherit from CommonException.

Example 14:

```
package org.csapi.jr.se;
public class ResourcesUnavailableException extends org.csapi.jr.se.CommonException {
        private Throwable _cause = null;

        public ResourcesUnavailableException () {
            super();
        }

        public ResourcesUnavailableException (String message) {
            super(message);
        }

        public ResourcesUnavailableException (String message, Throwable cause) {
            super(message, cause);
        }

        public ResourcesUnavailableException (Throwable cause) {
            _cause = cause;
        }


}
```

C.3.6.7.5 Additional abstract exceptions

Additional abstract exceptions (See ETSI ES 202 915-2, Annex D) have been defined which are TpInvalidArgumentException, TpFrameworkException, TpMobilityException, TpDataSessionException, TpMessagingException, TpConnectivityException, TpAccountException, TpPAMException and TpPolicyException and are mapped as follows:

Example 15:

```
package org.csapi.jr.se;
public class InvalidArgumentException extends java.lang.Exception {
        private Throwable _cause = null;

        public InvalidArgumentException () {
            super();
        }

        public InvalidArgumentException (String message) {
            super(message);
        }

        public InvalidArgumentException (String message, Throwable cause) {
            super(message);
            _cause = cause;
        }

        public InvalidArgumentException (Throwable cause) {
            _cause = cause;
        }

        public Throwable getCause() {
            return _cause;
        }
}
```

C.3.6.7.6 InvalidUnionAccessorException

An additional exception, InvalidUnionAccessorException, is defined which indicates that the expected data element has not been set.

Example 16:

```
package org.csapi.jr.se;
public class InvalidUnionAccessorException extends org.csapi.jr.se.InvalidArgumentException {
        private Throwable _cause = null;

        public InvalidUnionAccessorException (){
            super ();
        }

        public InvalidUnionAccessorException (String message){
            super (message);
        }

        public InvalidUnionAccessorException (String message, Throwable cause){
            super (message,cause);
        }

        public InvalidUnionAccessorException (Throwable cause) {
            _cause = cause;
        }

}
```

C.3.6.7.7 InvalidEnumValueException

An additional exception, InvalidEnumValueException, is defined which indicates that an enum data type was accessed with an invalid request value.

Example 17:

```
package org.csapi.jr.se;
public class InvalidEnumValueException extends org.csapi.jr.se.InvalidArgumentException {
        private Throwable _cause = null;

        public InvalidEnumValueException () {
            super ();
        }

        public InvalidEnumValueExceptions (String message) {
            super (message);
        }

        public InvalidEnumValueException (String message, Throwable cause) {
            super (message,cause);
        }

        public InvalidEnumValueException (Throwable cause) {
            _cause = cause;
        }

}
```

C.4 J2SE Specific Conventions

The UML interfaces are represented by Java public interfaces; those interfaces that inherit from other interfaces are represented in Java as extending that interface. The Java realisations of OSA/Parlay SCFs use an Event Listener design pattern while the Framework uses the Callback pattern.

This annex provides the information on realisation of the Java developer API including:

- How Java APIs are realised from Parlay UML

- Where the listener pattern is used, new classes to be generated from the UML
- Changes required to data types and methods to support correlation using object references
- Use of hierarchical exceptions

C.4.1 Removal of "Tp" Prefix

The UML data types labelled with the prefix “Tp” are represented in Java without this prefix.

Example 18:

UML	Java Realisation
TpCallAppInfo	CallAppInfo

In the case of name collisions between data types and interfaces as with IpTerminalCapabilities and IpService the UML data types labelled with the prefix “Tp” are represented in Java with an alternative prefix “Type”.

Example 19:

UML	Java Realisation
IpTerminalCapabilities	TerminalCapabilities
TpTerminalCapabilities	TypeTerminalCapabilities

The above example is based in conjunction with C.4.3 Removal of "Ip" Prefix.

C.4.2 Constants

The UML constants labelled with the prefix “P_” are represented in Java without this prefix.

Example 20:

UML Constant	Java Constant
P_NO_CALLBACK_ADDRESS_SET	NO_CALLBACK_ADDRESS_SET

C.4.3 Removal of "Ip" prefix

The "Ip" prefix is removed in the Java realisation of UML interfaces.

Example 21:

UML	Java
IpCallControlManager	CallControlManager

C.4.4 Mapping of IpInterface

IpInterface interface is represented by the CsapiInterface interface. This is a ‘marker’ interface, in that it contains no methods, but provides a common interface for related interfaces to inherit from. All interfaces to be serializable; this can be done by CsapiInterface extending Serializable.

Example 22:

```
package org.csapi.jr.se;
        public interface CsapiInterface extends Serializable{
        }
```

C.4.5 Mapping of IpService

IpService interface is represented by the Java Service interface. This provides a common interface for related interfaces to inherit from.

Example 23:

Service Interface:

```
package org.csapi.jr.se;
public interface Service extends CsapiInterface {
        public final static int IN_SERVICE_STATE=0 ;
        public final static int  OUT_OF_SERVICE_STATE=1;

        void addServiceStateChangeListener(ServiceStateChangeListener listener)
        int getServiceState();
        void removeServiceStateChangeListener( ServiceStateChangeListener listener) ;
}
```

Listener interface:

```
package org.csapi.jr.se;
public interface ServiceStateChangeListener extends java.util.EventListener {
        void onOutOfService(OutOfServiceEvent event);
}
```

Event class:

```
package org.csapi.jr.se;
public class OutOfServiceEvent extends  jav.util.EventObject {
    public OutOfServiceEvent(java.lang.Object source){
        super(source)
    }
}
```

C.4.6 Mapping of UML Operations

The UML operations are represented in Java as methods.

Exceptions that can be raised by UML operations are represented in Java with the throws clause and the Java Realisation of the UML Exceptions.

UML “in” parameters, represented by “in ” preceding the parameter type are represented in Java without this clause.

Example 24:

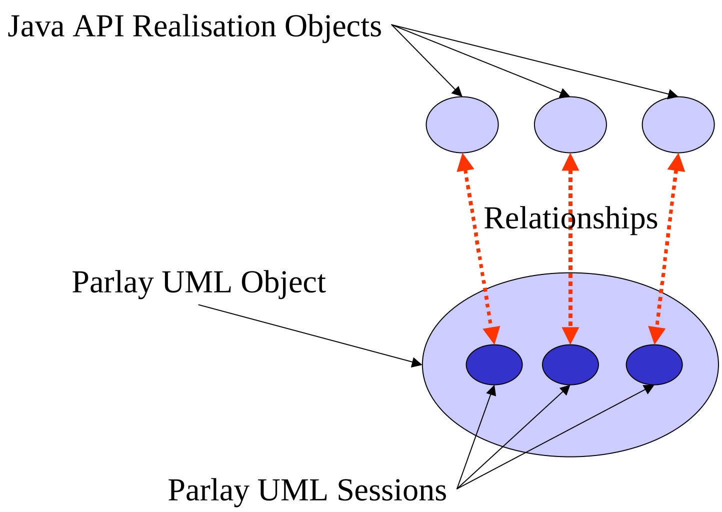
```
public void managerResumed ();

public CsapiInterface obtainInterface (InterfaceName interfaceName) throws
InvalidInterfaceNameException;

public Service createServiceManager (ClientAppID application, ServicePropertyList serviceProperties,
ServiceInstanceID serviceInstanceID);
```

The above example method signatures are based on generic mapping of interfaces, exceptions and data types.

C.4.7 Mapping of TpSessionID

The UML TpSessionID data types will be hidden in the J2SE APIs (and optionally supported by the underlying Java implementation). Consequently, the TpSessionIDSet data type and IpService.setCallbackWithSessionID() method are superfluous. Also, structures with only TpSessionID and interface references (e.g. TpCallIdentifier) are no longer necessary and references to these structures should be replaced by just the reference to the interface. For data types that contain TpSessionID the Java API Realisation object replaces theTpSessionID.

The following figure shows how Java API Realisation objects relate to Parlay UML objects and sessions. How this is realised in the adaptors is implementation dependent.



C.4.8 Mapping of TpAssignmentID to the creation of an Activity object.

The UML TpAssignmentID data types, which differentiate between multiple parallel asynchronous method invocations (activities) on the same ("parent") interface, are deleted and replaced with createXxx methods (one for each parallel asynchronous activity) that create ("child") activity interfaces. Where this would result in method names of the pattern createCreateXxx, this should be changed to method names with the pattern createXxx. Associated listeners would then remove the Create prefix from their name. These activity interfaces, in addition to possibly supporting other methods, will support one of the previously mentioned multiple parallel asynchronous method invocations. Hence, the Java API realisation creates multiple (activity) objects and invokes a single request per object rather than creating a single object and invoking multiple requests on that object, each request being differentiated using the TpAssignmentID value. The results of the asynchronous method invocation will be handled by the activity interface's listener interface. To create the activity interface, the original IpXxx interface (to be named Xxx) will replace its parallel supporting asynchronous method invocations, yyyYyyReq, with createYyyYyy methods that take no parameters but returns the activity interface, YyyYyy. Where this would result in method names of the pattern createCreateXxx, this should be changed to method names with the pattern createXxx. Associated listeners would then remove the Create prefix from their name. The activity interface will extend Activity interface (see next rule), have a simple FSM, the addYyyYyyListener, removeYyyYyyListener and the asynchronous method that previously supported a parallel capability (typically named yyyYyyReq, but also yyyYyyStop).

An Activity interface, packaged in org.csapi.jr.se, is added as a parent to all activity interfaces. An application may add listeners of type ActivityStateChangeListener to an Activity if it wishes be explicitly informed when the activity becomes invalid.

The YyyYyyListener activity listener interfaces will extend java.util.EventListener. The asynchronous methods of previously named IpAppXxx, typically labelled yyyYyyRes and yyyYyyErr but also yyyYyy, will be renamed onYyyYyyRes and onYyyYyyErr but also onYyyYyy. Each method will have an event parameter, typically labelled YyyYyyResEvent and YyyYyyErrEvent, but also YyyYyyEvent. Events will be classes that extend java.util.EventObject and contain a public constructor (with multiple parameters – one per class carried by the event) and a number of public getter methods (one per "gettable" class carried by the event). As a result of adding activity listener interfaces, this may cause the requirement for the original IpAppXxx to disappear, since the yyyYyyRes and yyyYyyErr methods will effectively be ported to the activity listener interfaces.

For data types that contain TpAssignmentID the activity object replaces the TpAssignmentID.

Example 25:

Activity Interface:

```
package org.csapi.jr.se;
public interface Activity  extends CsapiInterface {
        public final static int IDLE_STATE = 0;
        public final static int ACTIVE_STATE = 1;
        public final static int INVALID_STATE = 2;
        public int getState();
        public void addActivityStateChangeListener(ActivityStateChangeListener listener);
        public void removeActivityStateChageListener(ActivityStateChangeListener listener);
}
```

Activity Listener Interface and Event class:

```
package org.csapi.jr.se;
public interface ActivityStateChangeListener {
        onInvalidStateEvent (InvalidActivityEvent event)
}

public class InvalidActivityEvent extends java.util.EventObject {
    public InvalidActivityEvent(java.lang.Object source){
        super(source)
    }

}
```

Parent interface:

```
package org.csapi.jr.se.mmm.ul;
public interface UserLocation extends org.csapi.jr.se.Service {
        public LocationReport createLocationReport();
        public ExtendedLocationReport createExtendedLocationReport();
        public PeriodicLocationReporting createPeriodicLocationReporting();
}
```

Child Interface:

```
package org.csapi.jr.se.mm.ul;
public interface LocationReport extends org.csapi.jr.se.Activity {
        public void addLocationReportListener(LocationReportListener listener)
        public void removeLocationReportListener(LocationReportListener listener)
        public void locationReportReq(Address[] users) throws …
}
```

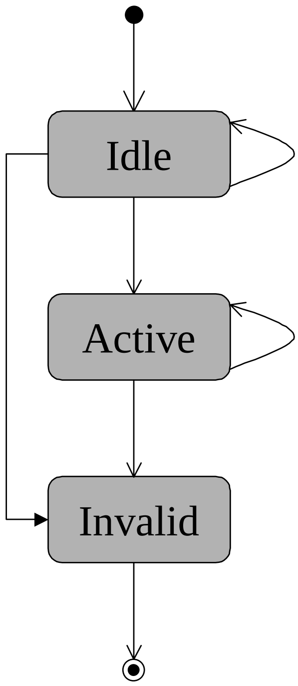
Listener Interface:

```
package org.csapi.jr.se.mm.ul;
public interface LocationReportListener extends java.util.EventListener {

        public void onLocationReportResEvent(LocationReportResEvent event);
        public void onLocationReportErrEvent(LocationReportErrEvent event);
}
```

Event classes:

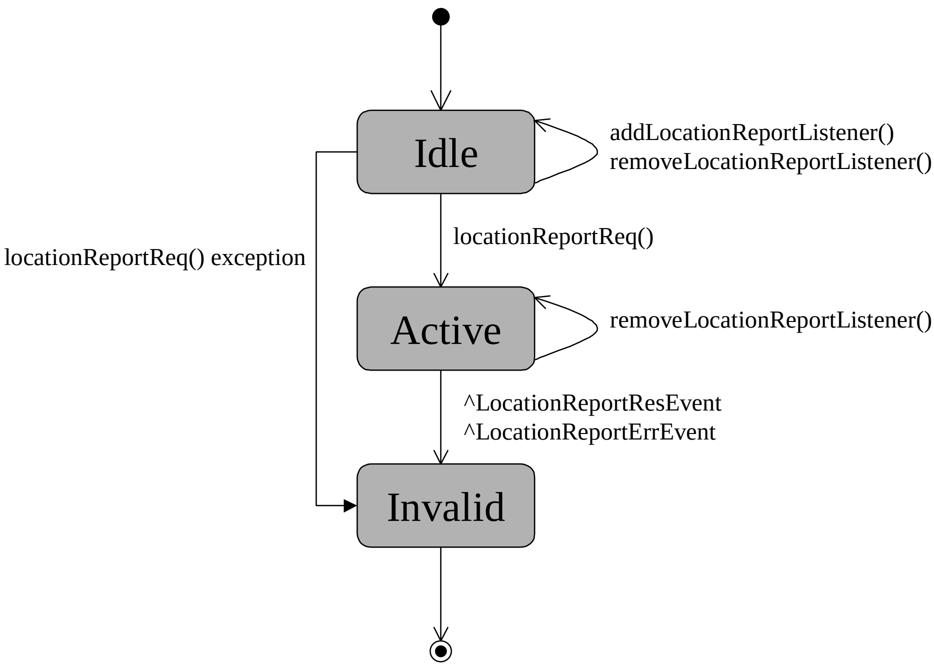
```
package org.csapi.jr.se.mmm.ul;
public class LocationReportResEvent extends java.util.EventObject{
        // with a public UserLocation[] constructor and a public getter
        // method for the parameter of the event
}

public classLocationReportErrEvent extends java.util.EventObject {
        // with a public MobilityError and MobilityDiagnostic constructor
        // and two public getter methods, one for each of the parameters
        // of the event
}
```

The Finite State Model for the Activity interface is given below:

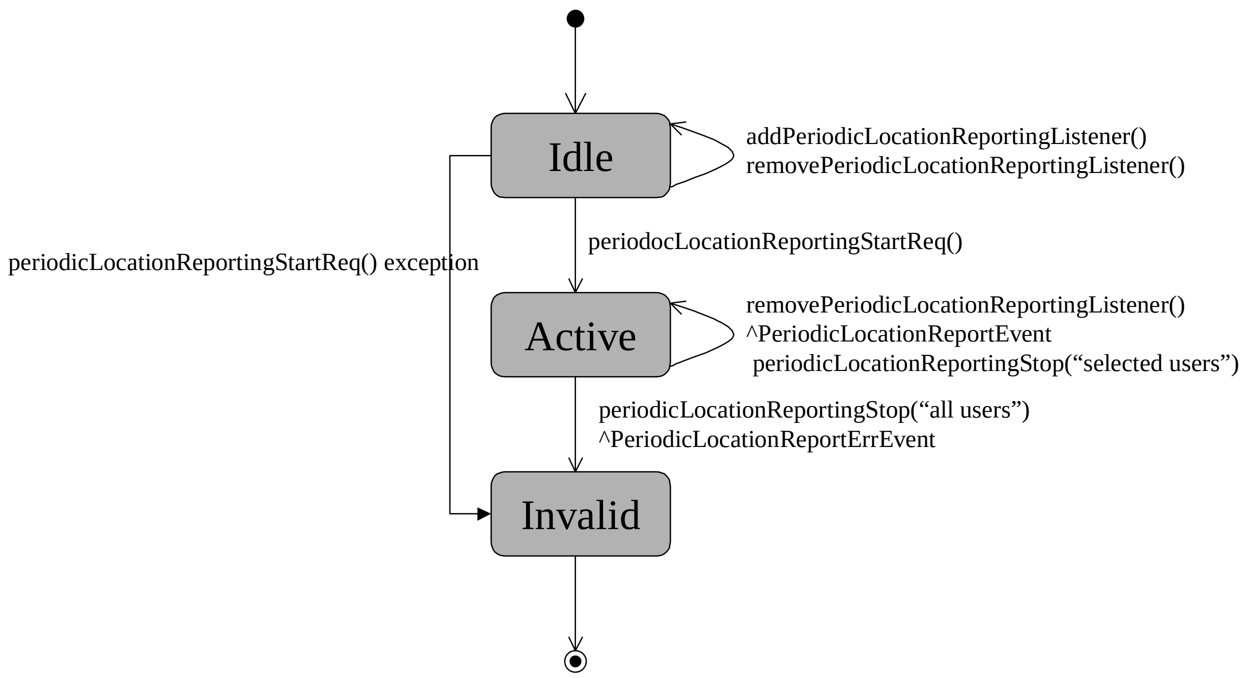


This interface specifies an activity, which might be provided by a service. An activity has three states: "idle", "active" and "invalid". The initial state is "idle" and here the listeners should be registered. It performs in the "active" state. It enters the "invalid" state when it has fulfilled its task or a fatal error occurred. In special cases state transition from "idle" to "invalid" is possible.

An example activity interface FSM is given below for a single activity request with a single response:



An example activity interface FSM is given below for a single activity request with repeating responses:



C.4.9 Callback Rule

The UML callback design pattern for non client-to-service interfaces (Parlay interface numbers 1, 3, 4, 5 and 6 [Fig 1]) is represented in Java with the callback design pattern. The UML callback design pattern for client-to-service interfaces (Parlay interface number 2 [Fig 1]) is represented in Java with the event listener design pattern.

The UML client-to-service interfaces (Parlay interface number 2) with the IpAppXxxx naming convention are represented in Java with the XxxxListener naming convention.

The IpService.setCallback method can be deleted; the interfaces that inherited the setCallback method now have associated addXxxxListener and removeXxxxListener methods. According to the *TpSessionID* mapping, IpService.setCallbackWithSessionID() method is deleted.

The XxxxListener listener interfaces will extend java.util.EventListener. The asynchronous methods of previously named IpAppXxxx, typically labelled yyyyYyyyRes and yyyyYyyyyErr but also yyyyYyyy, will be renamed onYyyyYyyyRes and onYyyyYyyyErr but also onYyyyYyyy. Each method will have an event parameter, typically labelled YyyyYyyyResEvent and YyyyYyyyErrEvent, but also YyyyYyyyEvent. Events will be classes that extend java.util.EventObject and contain a private constructor (with multiple parameters – one per class carried by the event) and a number of public getter methods (one per "gettable" class carried by the event). Events are read-only and serializable.

Example 26:

Listener Interface:

```
package org.csapi.jr.se.cc.mpccs;

MultiPartyCallListener extends java.util.EventListener{

public void onGetInfoResEvent(GetInfoResEvent event)
public void onGetInfoErrEvent(GetInfoErrEvent event)
public void onSuperviseResEvent(SuperviseResEvent event)
public void onSuperviseErrEvent(SuperviseErrEvent event)
public void onCallEndedEvent(CallEndedEvent event)
public void onCreateAndRouteCallLegErrEvent(CreateAndRouteCallLegErrEvent event)
}
```

MuliPartyCall Interface additional methods:

```
public void addMultiPartyCallListener(MultiPartyCallListener multiPartyCallListener);
```

```
public void removeMultiPartyCallListener(MultiPartyCallListener multiPartyCallListener);
```

C.4.10 Factory Rule

The following Factory class allows applications to obtain proprietary peer API objects. The term "peer" is Java nomenclature for a particular platform-specific implementation of a Java interface.

Example 27:

```
package org.csapi.jr.se.fw;
import org.csapi.jr.se.PeerUnavailableException;
import org.csapi.jr.se.InvalidArgumentException;
import org.csapi.jr.se.ResourcesUnavailableException;
import org.csapi.jr.se.fw.access.tsm.Initial;
import java.util.*;

public class InitialFactory {
        private static InitialFactory myFactory;
        private static String className = null;
        private static String lang     = "en";
        private static String cntry    = "US";

        private InitialFactory() {
        }

        public synchronized Initial createInitial(String initialPeerReference) throws
PeerUnavailableException, ResourcesUnavailableException , InvalidArgumentException {
            Locale currentLocale;
            ResourceBundle messages;
            String tryMessage;

            try {
                currentLocale = new Locale(lang, cntry);
                messages = ResourceBundle.getBundle("InitialFactoryBundle", currentLocale);

                // Validate all used values before using them later
                // avoiding error text exception to hide the real exception

                tryMessage = messages.getString("InitialPeerReferenceNull");
                tryMessage = messages.getString("InitialInstFailure");
                tryMessage = message.getString("DestroyInitialFailure");
            }
            catch (Exception e) {
                throw new ResourcesUnavailableException ("Localisation failed to be initialized");
            }

            if (initialPeerReference == null) {
                String errmsg = messages.getString("InitialPeerReferenceNull");
                throw new InvalidArgumentException (errmsg);
            }

            try {
                Class c = Class.forName (getImplementationClassName ());
                if(initialPeerReference.equals(“”)){
                    // Creates a new instance of the Object class
                    // using default constructor
                    return (Initial)c.newInstance ();
                }

                Class[] paramTypes = {initialPeerReference.getClass()};
                java.lang.reflect.Constructor ctor =
                c.getConstructor(paramTypes);
                Object[] params = {initialPeerReference};
                return (Initial) ctor.newInstance(params);
            } catch (Exception e) {
                String errmsg = messages.getString("InitialInstFailure");
                throw new PeerUnavailableException (errmsg);
            }
        }

        public synchronized static InitialFactory getInstance() {
            if (myFactory == null) {
                myFactory = new InitialFactory ();
            }
```

```
            return myFactory;
        }

        public String getImplementationClassName () {
            return className;
        }

        public static void setImplementationClassName (String className) {
            this.className = className;
        }

        public synchronized static void setLocale(String language, String country) {
            if (langauage == null) {
                lang = "en";
            }
            else {
                lang = language;
            }

            if (country == null) {
                cntry = "US";
            }
            else {
                cntry = country;
            }
        }

        public void destroyInitial(Initial initialInstance) {
            if (initialInstance == null) {
                return;
            }

            try {
                delete initialInstance;
            } catch (Exception e) {
                String errmsg = messages.getString("DestroyInitialFailure");
                throw new RuntimeException(errmsg);
            }
        }
}
```

C.4.11 J2SE Specific Exceptions

Exceptions in this section are only applicable within a J2SE environment.

C.4.11.1 PeerUnavailableException

PeerUnavailableException indicates failure to access an implementation of the Initial interface.

Example 28:

```
public class PeerUnavailableException extends java.lang.Exception {
        private Throwable _cause = null;
        public PeerUnavailableException () {
            super();
        }

        public PeerUnavailableException (String message) {
            super(message);
        }

        public PeerUnavailableException (String message, Throwable cause) {
            super(message);
            _cause = cause;
        }

        public PeerUnavailableException (Throwable cause) {
            _cause = cause;
        }

        public Throwable getCause() {
            return _cause;
        }
```

```
}
```

C.4.11.2 IllegalStateException

IllegalStateException exception signals that a method has been invoked at an illegal or inappropriate time.

Example 29:

```
package org.csapi.jr.se;
public class IllegalStateException extends java.lang.Exception {

        private int _state;
        private  java.lang.Object _object;

        public IllegalStateException(Object object, int state) {
            super();
            _object = object;
            _state  = state;
        }

        public Illegal StateException(Object object, int state, String s) {
            super(s);
            _object = object;
            _state  = state;
        }

        public Object getObject() {
            return _object;
        }

        public int getState() {
            return _state;
        }
}
```

C.4.12 User Interaction Specific Rules

C.4.12.1 Interfaces representing UML IpUI and IpUICall Rule

The following mappings take account of the fact that when the TpAssignmentID rule is applied the Java interfaces representing UML IpUICall does not extend the Java interfaces representing UML IpUI.

Java UIGeneric replaces the UML IpUI. Methods common to both the Java UIGeneric and Java UICall are pulled up into a super-interface called UI. UML IpAppUI and IpAppUiCall interfaces are replaced by a UIListener interface.

C.4.12.2 Naming Collisions of GUI and CUI Activities Rule

Naming collisions that arise through GUI and CUI activities e.g. XXX, having the same name will be dealt with by prefixing the Call Related UI activity by “CallRelated”. Methods to create the activity will become createCallRelatedXXX().

C.5 J2EE Specific Conventions

J2EE supports both remote and local interfaces.

C.5.1 Serialization UID

All data types will have a serialVersionUID defined within its definition, as a static final long value.

Example 30:

```
package org.csapi.jr.ee;
public final class TpAddress implements java.io.Serializable {
        static final long serialVersionUID = 989898989898L;

        private TpAddressPlan plan;
        … remainder of class …
}
```

C.5.2 Remote Interface Definitions

C.5.2.1 IpInterface

This interface implements java.io.Serializable. Since it is the root interface for all other interfaces, this makes all defined interfaces serializable.

Example 31:

```
public interface IpInterface extends java.io.Serializable
```

C.5.2.2 Methods for Remote Interfaces

A public method is defined within a remote interface for each method defined in the specification, with zero or one output specified as the return value, and all other parameters listed without any input marker. Each method will return java.rmi.RemoteException in addition to other exceptions, if any.

Example 32:

```
public void deassignCall (int callSessionID) throws java.rmi.RemoteException,
org.csapi.jr.ee.TpCommonException, org.csapi.jr.ee.InvalidSessionIdException;
```

C.5.3 Local Interface Definitions

C.5.3.1 Methods for Local Interfaces

A public method is defined within a local interface for each method defined in the specification, with zero or one output specified as the return value, and all other parameters listed without any input marker.

Example 33:

```
public void deassignCall (int callSessionID) throws org.csapi.jr.ee.TpCommonExceptions,
org.csapi.jr.ee.InvalidSessionIdException;
```

C.5.4 Multi Party Call Control Specific Rules

The Multi Party Call Control Manager interface has specific Java Realisation considerations.

C.5.4.1 IpCallLeg and IpAppCallLeg method name conflicts

Some method names within the IpAppCallLeg interface have the same names as methods in the IpAppMultiPartyCall interface. These method names conflict when both interfaces are implemented on the same object within an RMI/IIOP or CORBA environment.

For the method names that are the same in both IpMultiPartyCall and IpCallLeg interfaces or IpAppMultiPartyCall and IpAppCallLeg, the call leg related method names are modified to include "CallLeg" as part of the method name to avoid name conflicts. The following method names result:

IpCallLeg Method Name	Realisation Method Name
getInfoReq	getCallLegInfoReq
superviseReq	superviseCallLegReq

Table 1: IpCallLeg method name modifications

IpAppCallLeg Method Name	Realisation Method Name
getInfoRes	getCallLegInfoRes
getInfoErr	getCallLegInfoErr
superviseRes	superviseCallLegRes
superviseErr	superviseCallLegErr

Table 2: IpAppCallLeg method name modifications

Annex D (informative): Change history

Change history							
Date	**TSG #**	**TSG Doc.**	**CR**	**Rev**	**Subject/Comment**	**Old**	**New**
Mar 2001	CN_11	NP-010134	047	--	CR 29.198: for moving TS 29.198 from R99 to Rel-4 (N5-010158)	3.2.0	4.0.0
Jun 2001	CN_12	NP-010330	001	--	Corrections to OSA API Rel4 (Correction to IDL namespace to align with that of ETSI and Parlay equivalent APIs: Change org.open_service_access root namespace to org.csapi) (N5-010267)	4.0.0	4.1.0
Sep 2001	CN_13	NP-010464	002	--	Changing references to JAIN	4.1.0	4.2.0
Dec 2001	CN_14	NP-010594	003	--	Replace Out Parameters with Return Types	4.2.0	4.3.0
Dec 2001	CN_14	NP-010594	004	--	Remove the perception that the OSA API only uses CORBA for its transport mechanism	4.2.0	4.3.0
Mar 2002	--	--	--	--	Editorial update (no CR) following Hong Kong CN5#16	4.3.0	4.3.1
Jun 2002	CN_16	NP-020181	005	--	Addition of support for Java API technology realisation	4.3.1	5.0.0
Jun 2002	CN_16	NP-020182	006	--	Addition of support for WSDL realisation	4.3.1	5.0.0
Jun 2002	CN_16	NP-020184	007	--	Adding the full naming convention for exceptions	4.3.1	5.0.0
Jun 2002	CN_16	NP-020184	008	--	Correction of References in OSA specifications	4.3.1	5.0.0
Jun 2002	CN_16	NP-020184	009	--	Addition of text describing the technology realisations of the Parlay/OSA specification	4.3.1	5.0.0
Sep 2002	CN_17	NP-020427	010	--	Addition to ObjectRef description in WSDL Mapping Rules	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	011	--	Addition of sequence tag to Choice types	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	012	--	Replace all occurrences of the xsd:anyURI type to xsd:string	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	013	--	Correction to Namespace mapping in WSDL Mapping Rules	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	014	--	Correction to xmlns:wsdl Namespace	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	015	--	Prepend class name to <message> name.	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	016	--	Correction to void return types in WSDL Mapping Rules	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	017	--	Add missing CORBA realization rules in Part 1	5.0.0	5.1.0
Sep 2002	CN_17	NP-020427	018	--	Add general introduction to the OSA APIs in Part 1	5.0.0	5.1.0
Sep 2002	CN_17	NP-020395	020	--	Add text to clarify relationship between 3GPP and ETSI/Parlay OSA specifications	5.0.0	5.1.0
Mar 2003	CN_19	--	--	--	Editorial update (no CR) following Bangkok CN5#22 (Introduction, Reference Titles)	5.1.0	5.1.1
Jun 2003	CN_20	NP-030298	022	1	Removal of un-used references	5.1.1	5.2.0
Jun 2003	CN_20	NP-030239	023	--	Correction to Java Realisation Annex	5.1.1	5.2.0
Sep 2003	CN_21	NP-030352	024	--	Correction to Java Realisation Annex	5.2.0	5.3.0
Dec 2003	CN_22	NP-030547	025	--	Add Java Realization rules to solve MPCC name conflicts	5.3.0	5.4.0
Dec 2003	CN_22	NP-030547	026	--	Correction to Java Realisation Rulebook	5.3.0	5.4.0

History

Document history		
V5.0.0	June 2002	Publication
V5.1.0	September 2002	Publication (Withdrawn)
V5.1.1	March 2003	Publication
V5.2.0	June 2003	Publication
V5.3.0	September 2003	Publication
V5.4.0	December 2003	Publication