INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Series Q

Supplement 28
(12/1999)

SERIES Q: SWITCHING AND SIGNALLING

Technical Report: Signalling and Protocol Framework for an Evolving Environment (SPFEE) – Specifications for service access

ITU-T Q-series Recommendations – Supplement 28

(Formerly CCITT Recommendations)

ITU-T Q-SERIES RECOMMENDATIONS

SWITCHING AND SIGNALLING

SIGNALLING IN THE INTERNATIONAL MANUAL SERVICE	Q.1–Q.3
INTERNATIONAL AUTOMATIC AND SEMI-AUTOMATIC WORKING	Q.4–Q.59
FUNCTIONS AND INFORMATION FLOWS FOR SERVICES IN THE ISDN	Q.60–Q.99
CLAUSES APPLICABLE TO ITU-T STANDARD SYSTEMS	Q.100–Q.119
SPECIFICATIONS OF SIGNALLING SYSTEMS No. 4 AND No. 5	Q.120–Q.249
SPECIFICATIONS OF SIGNALLING SYSTEM No. 6	Q.250–Q.309
SPECIFICATIONS OF SIGNALLING SYSTEM R1	Q.310–Q.399
SPECIFICATIONS OF SIGNALLING SYSTEM R2	Q.400–Q.499
DIGITAL EXCHANGES	Q.500–Q.599
INTERWORKING OF SIGNALLING SYSTEMS	Q.600–Q.699
SPECIFICATIONS OF SIGNALLING SYSTEM No. 7	Q.700–Q.849
DIGITAL SUBSCRIBER SIGNALLING SYSTEM No. 1	Q.850–Q.999
PUBLIC LAND MOBILE NETWORK	Q.1000–Q.1099
INTERWORKING WITH SATELLITE MOBILE SYSTEMS	Q.1100–Q.1199
INTELLIGENT NETWORK	Q.1200–Q.1699
SIGNALLING REQUIREMENTS AND PROTOCOLS FOR IMT-2000	Q.1700–Q.1799
BROADBAND ISDN	Q.2000–Q.2999

For further details, please refer to the list of ITU-T Recommendations.

Supplement 28 to ITU-T Q-series Recommendations

Technical Report: Signalling and Protocol Framework for an Evolving Environment (SPFEE) – Specifications for service access

Summary

This Supplement specifies the (Session level) information and computational model of the Consumer-Retailer reference point. It provides information and computational specification for service access defined in Signalling and Protocol Framework for an Evolving Environment (SPFEE). The information and computational specifications are described in informal and formal description languages such as Interface Definition Language (IDL).

Source

Supplement 28 to ITU-T Q-series Recommendations was prepared by ITU-T Study Group 11 (1997-2000) and approved under the WTSC Resolution 5 procedure on 3 December 1999.

Keywords

Access, computational model, IDL, information model, ODP, reference point, session.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSC Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this publication, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this publication may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the publication development process.

As of the date of approval of this publication, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this publication. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

Supplement 28 to ITU-T Q-series Recommendations

Technical Report: Signalling and Protocol Framework for an Evolving Environment (SPFEE) – Specifications for service access

(Geneva, 1999)

1 Scope

This Supplement provides:

– information specification; and

– computational specification

for service access defined in Signalling and Protocol Framework for an Evolving Environment (SPFEE). These include the Consumer-Retailer Reference Point specifications. The information and computational specifications are described in informal and formal description languages such as Interface Definition Language (IDL).

2 References

The following Technical Reports and other references contain provisions which, through reference in this text, constitute provisions of this Supplement. At the time of publication, the editions indicated were valid. All Supplements and other references are subject to revision; all users of this Supplement are therefore encouraged to investigate the possibility of applying the most recent edition of the supplements and other references listed below. A list of the currently valid ITU-T Recommendations and supplements is regularly published.

[1] ITU-T Recommendation X.901 (1997) | ISO/IEC 10746-1:1998, *Information technology – Open distributed processing – Reference model: Overview*.

[2] ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, *Information technology – Open distributed processing – Reference Model: Foundations*.

[3] ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, *Information technology – Open distributed processing – Reference Model: Architecture*.

[4] ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1999, *Information technology – Open distributed processing – Interface Definition Language*.

[5] ITU-T Recommendation Z.130 (1999), *ITU object definition language*.

[6] ITU-T Q-series Recommendations – Supplement 27 (1999), *Technical Report*: *Overview of Signalling and Protocol Framework for an Evolving Environment (SPFEE)*.

3 Definitions

This Supplement defines the following terms in addition to those in [6]:

3.1 Domain Access Session (D_AS): An abstract object which represents the generic information required to establish and support access between two domains.

3.2 User Domain Access Session (UD_AS): An object managed by the user and representing the collection of capabilities and configuration that the user employs to contact a provider.

3.3 Provider Domain Access Session (PD_AS): An object managed by the provider and created when the user becomes a recognized, identifiable entity with specific capabilities and data within the provider domain.

3.4 Provider Service Session (PSS): A central view of the service session, including all members and any additional provider information and logic necessary to execute service requests and maintain the session.

3.5 Usage Service Session (USS): A session member's (e.g. an end-user's) customized view of a service.

3.6 Domain Usage Service Session (D_USS): An abstract object which represents generic information required to establish and support a service between two domains for an associated session member.

3.7 User Domain Usage Service Session (UD_USS): Functionality and information present in an end-user domain, e.g. a consumer domain, to support the usage service session and allow the end-user to interact with the service.

3.8 Provider Domain Usage Service Session (PD_USS): Functionality and information present in a provider domain (e.g. retailer, third party provider) to support the usage service session in a usage provider role.

3.9 Domain Usage Service Session Binding (D_USS_Binding): Dynamic information associated with binding two D_USSs.

4 Abbreviations

This Supplement uses the following abbreviations:

3Pty	Third-party inter-domain reference point
anonUA	Anonymous User Agent
as-UAP	User Application (Access Session related)
AS	Access Session
Bkr	Broker inter-domain reference point
CO	Computational Object
ConS	Connectivity Service inter-domain reference point
CS	Communication Session
CSLN	Client-server layer inter-domain reference point
DPE	Distributed Processing Environment
FCAPS	Fault, Configuration, Accounting, Performance, Security
IA	Initial Agent
IDL	Interface Definition Language
LNFed	Layer network federation inter-domain reference point
NamedUA	Named User Agent
ODL	Object Definition Language
ODP	Open Distributed Processing
OMT	Object Modelling Technique
PA	Provider Agent

Ret	Retailer inter-domain reference point
RP	Reference point
RtR	Retailer to Retailer inter-domain reference point
SC	Service Component
SF	Service Factory
SPFEE	Signalling and Protocol Framework for an Evolving Environment
SS	Service Session
SSM	Service Session Manager
ss-UAP	User Application (Service Session related)
Tcon	Terminal Connection inter-domain reference point
UA	User Agent
UAP	User APplication
USM	User Service Session Manager

5 Information Model of the Session and Resource Level

The information model is described in the OMT object model notation. Typical symbols are illustrated in Figure 5-1. The abstract classes (for which there is no object instance) are shown in 'curly bracket' style[1]: they have no other purpose but to simplify the object oriented modelling.

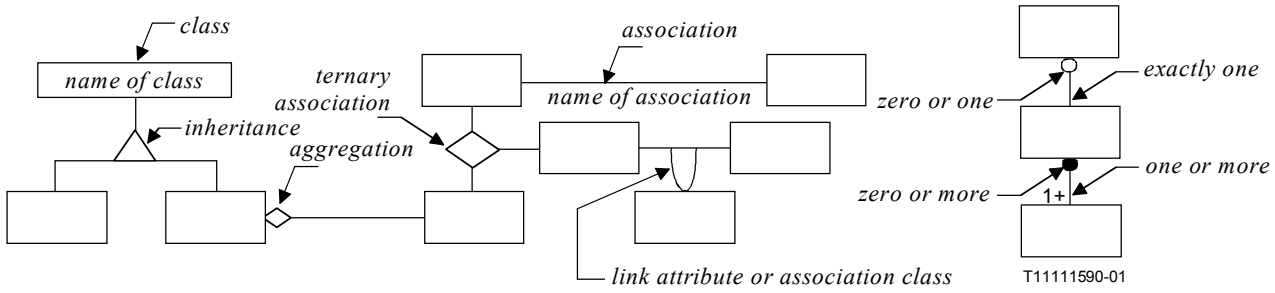


Figure 5-1 – Typical symbols in an OMT diagram

[1] I.e. '{}' is inserted in front of the class name in the diagrams.

5.1 Sessions, Services and Domains

This subclause presents the information model as related to session related services[2]. As a session related service may extend over multiple domains, which are treated as business administrative domains, it is useful to model the service as an aggregation of one or more domain services, where each domain service represents a part of a service confined to a single domain. A session is an instance of a service. Just as a session may represent an instance of a service, a domain session represents an instance of a domain service. Domain sessions may interact to establish services extending over multiple domains.

Figure 5-2 shows the relationship between domain, service and session objects. A session related service is an aggregation of other session related services and domain services. Each domain service is instantiated by one domain session, which is established in and managed by its associated domain. A domain session may be bound to another domain session via a domain session binding, which represents the dynamic information used to link the two sessions.

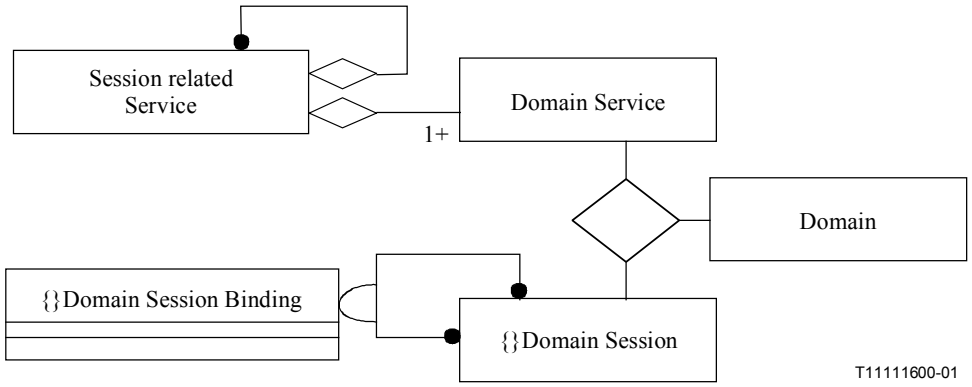


Figure 5-2 – Session Information Model

5.2 Classification of Sessions

This subclause considers the classification of sessions. A number of classification schemes are possible, but they are all based on the session-root object. This generic object defines the common properties of a session. All derived objects, regardless of their classification, inherit these common attributes and operations, such as: session identifier, session type, state, terminate, suspend, and resume.

A classification is based on the service (functional) separations of access, service and communication. Sessions have been identified to support each of these separations. Though specialized to support the particular requirements of a functional area, each session retains the common properties of the session-root object.

It is also helpful to use the domain session and domain session binding to help classify sessions. Again, the session-root object is the root of the hierarchy[3], and the domain session and domain

---

[2] A service is classified into two types of services: one is 'session related services', and the other is 'non-session related services'. 'Non-session related services' are beyond the scope of this Supplement. The session related services are defined as follows: Session related service represents an online telecommunication service which is offered to users **with** telecommunication equipment and resources. For example, all telecommunication services, from POTS to multimedia services and/or Internet related services, are categorized as session related service.

[3] The session-root object is an abstract object which means this object is not instantiable.

session binding inherit its properties. This type of classification can be combined with the previous scheme to classify all the session objects for each separation. The following sections will use this combined scheme and the following aspects to classify objects:

– User aspects: They represent the entities, semantics, constraints and rules governing the availability and usage of service capabilities with respect to a specific user, and the resources and mechanisms needed to actually support the service capabilities according to a specific user standpoint;

– Provider aspects: They represent the entities, semantics, constraints and rules governing the provision and usage of service capabilities to users, as well as the resources and mechanisms needed to actually support those capabilities.

5.3 Classification of Access Sessions

Access sessions can be classified in terms of a specialization hierarchy as shown in Figure 5-3. The classification is along the separation into user and provider aspects.

Relating this to Figure 5-2, the *Domain Access Session* corresponds (by subclassing) to the domain session, and the *Access Session* corresponds (by subclassing) to the domain session binding.

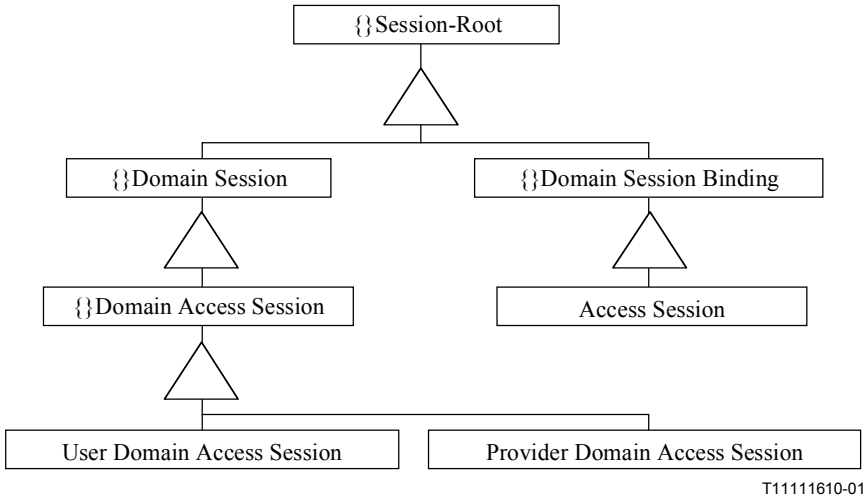


Figure 5-3 – Classification of the Access Session

5.4 Access Session

The access related information objects and their relationships are shown in Figure 5-4.

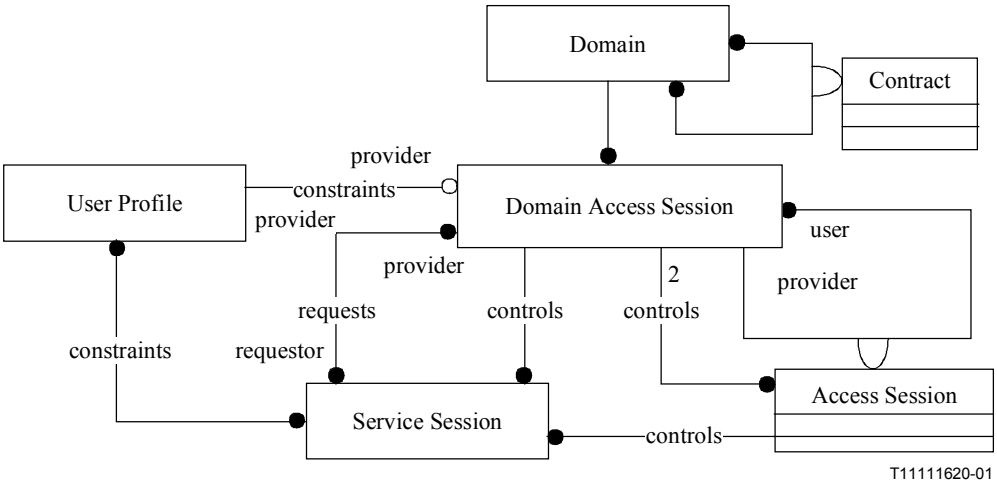


Figure 5-4 – Relationships between Access Related Information Objects

Each domain access session is associated with a particular access role. Two roles have been identified: user and provider. The user role accepts invitations and makes requests on the associated domain access session. The provider role accepts requests and sends invitations to the associated domain access session.

The access session is categorized by the roles supported by each domain access session. These roles depend on the relation between domains. The following access session types may be required:

– **Asymmetric type access session**: One domain acts in the user role and the other domain acts in the provider role;

– **Symmetric type access session**: Supporting symmetry in the access session, both domains support both roles.

The information objects and their relationships for the asymmetric type access session are shown in Figure 5-5. Figure 5-5 presents a more detailed version of Figure 5-4 for the asymmetric type access session. An asymmetric session is supported by the User Domain Access Session and the Provider Domain Access Session.

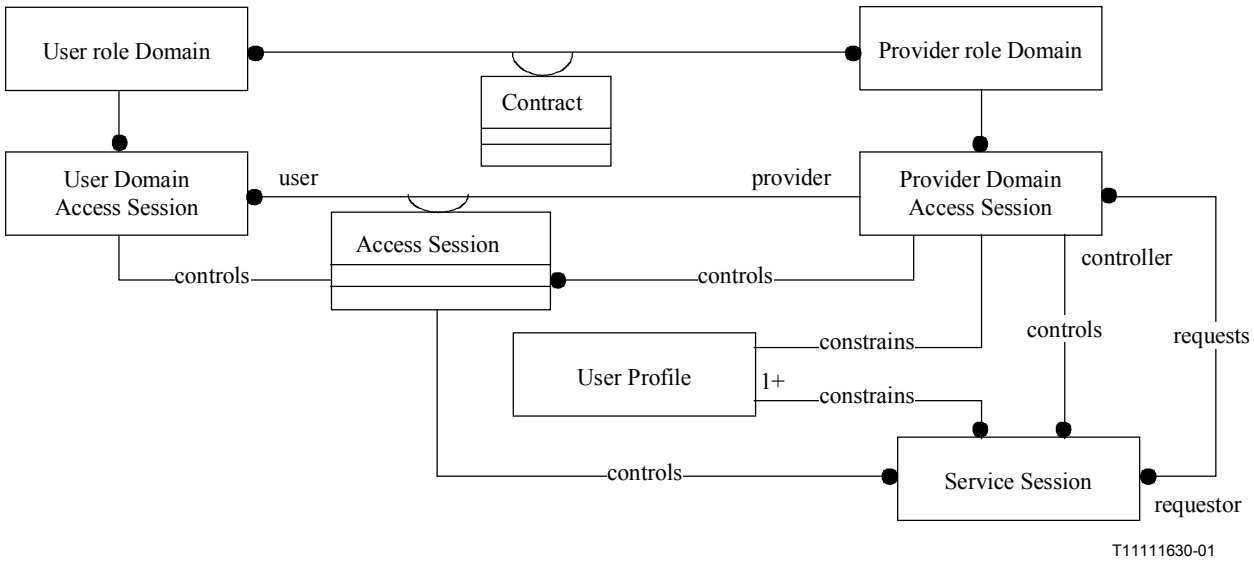


Figure 5-5 – Information objects and their relationship for asymmetric type access sessions

5.4.1 Domain Access Session (D_AS)

This is an abstract object which represents the generic information required to establish and support access between two domains. Each domain access session is associated with a particular domain. However, a domain may have many domain access sessions. A domain access session is usually associated with one (or possibly more) contractual relations with another domain. There may be multiple domain access sessions within the domain for each contractual relationship.

The D_AS is specialized into user domain access session and provider domain access session type informational objects. They are all described in the following subclauses.

5.4.1.1 User Domain Access Session (UD_AS)

This object is managed by the user and represents the collection of capabilities and configuration that the user employs to contact a provider. The UD_AS holds user defined policies that determine the terms of the interaction with a provider, such as security policy and accounting; they form the basis of the negotiation with a provider in the establishment of a mutually acceptable access session. The UD_AS may comprise one to many terminals or DPE nodes. Whatever the UD_AS configuration, both the user and provider will have a perspective on how trustworthy the UD_AS is. This trust level (e.g. confidentiality, password protection, cipher implementations) will be reflected in the management restrictions imposed on the access session (e.g. refusal of high value services). For example, a UD_AS supported on a tamper resistant terminal supplied by the provider is more trustworthy than a small multi-user PC LAN.

5.4.1.2 Provider Domain Access Session (PD_AS)

This object is managed by the provider and can be considered to be created when the user becomes a recognized, identifiable entity with specific capabilities and data within the provider domain. The session terminates when the user ceases to have any user relationship with the provider (so the provider stops holding permanent information about the user in the User Profile). This object knows persistent information about the user. Part of this information specifies policies that determine the terms of interaction with the user, such as security policy and accounting. These policies form the basis of the negotiation with the user in the establishment of a mutually acceptable access session. Both the user and retailer will have a perspective on how trustworthy the PD_AS is. This trust level (e.g. confidentiality, password protection, cipher implementations) will be reflected in the management restrictions imposed on the access session.

5.4.2 Access Session (AS)

This object is managed by the user or the provider via the D_AS, and represents the collection of dynamic information for a binding between D_ASs, such as security policy, accounting and session description for this binding. This object terminates when the user or the provider ends the relationship (dynamic binding between D_ASs) with the provider or the user.

5.4.3 User Profile

The User Profile contains all information that is used directly by the D_AS for authorization decisions, constraints and customization of the D_ASs, Access Sessions (within Access Session Bindings) and Service Sessions.

5.5 Classification of Service Sessions

Figure 5-6 gives the classification hierarchy of service sessions along the separation into user and provider aspects.

Relating this to Figure 5-2, the Provider Service Session (PSS) and the Domain Usage Service Session (D_USS) correspond (by subclassing) to the domain session, and the domain usage service session bindings corresponds (by subclassing) to the domain session binding.

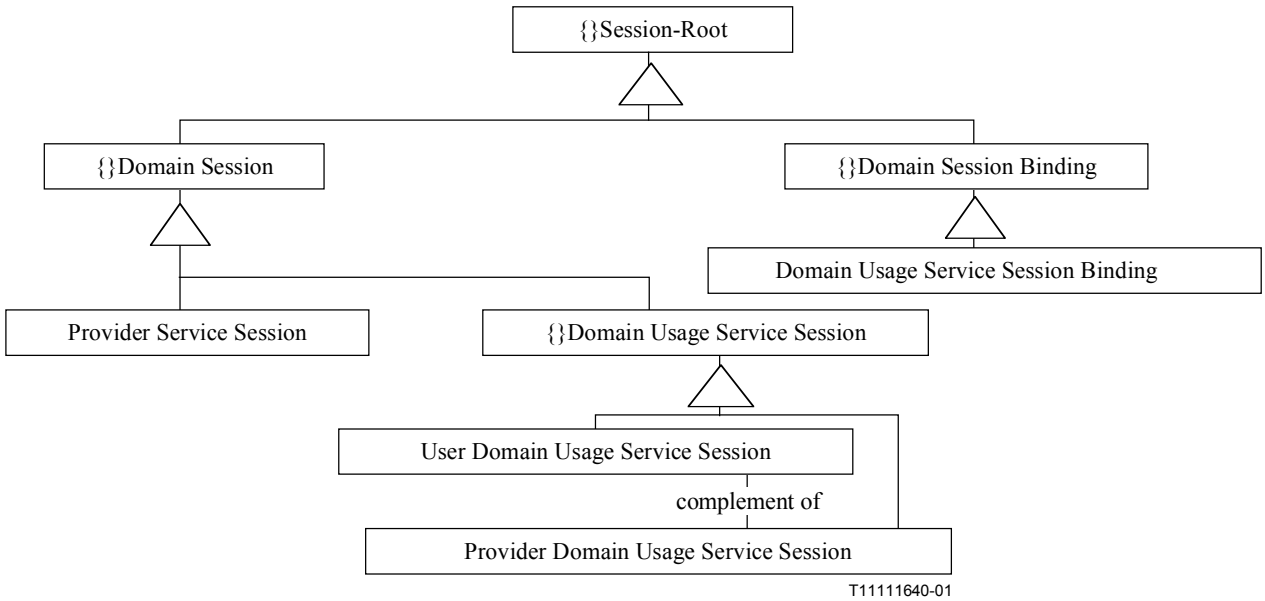


Figure 5-6 – Classification of the service session

5.6 Service Session

The service session related objects and their relationships are shown in Figure 5-7.

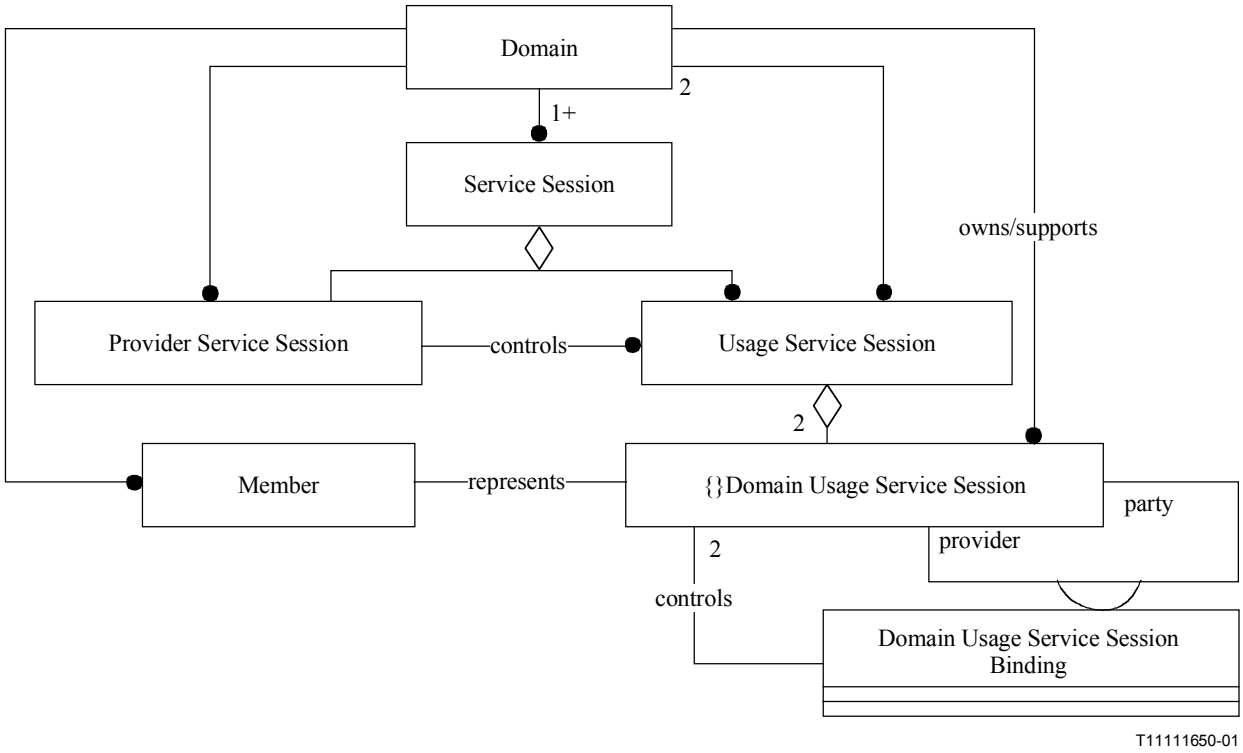


Figure 5-7 – Service session related objects and their relationships

The service session consists of usage and provider service sessions. Each usage service session can extend over two domains and is composed of two complementary Domain Usage Service Sessions (D_USSs), where a complementary D_USS is one that can interact with another. Each member of a session, i.e. an end-user, service resource, or associated session, is associated with a usage service session.

The type of D_USS supported depends on the perceived role of that member in the service session. Figure 5-6 shows the possible D_USSs. The following generic usage roles are supported:

– Usage Party: Resembles an "end-user" of a service, an active role which may make requests and is sent notifications of session changes.

– Usage Provider: Provides a service or acts as a service resource to another entity. This is a passive role in the sense that it cannot initiate actions, but responds to requests or notifications of changes.

More roles are possible. These include service specific roles, which are outside of the scope, and control and management roles to support relationships in composed services.

When a service session is started, or when a new member joins a service session, it acquires the relevant user profile information from the access session or domain access session (e.g. service description) for the member. This constrains the usage service session and potentially the provider service session. In the case of multi-member service sessions, an individual's user profile or current usage configuration may affect the whole service session, depending on the nature of the service and its management policies.

A service session can be instantiated by an access session, domain access session, or another service session. The initiator of a service session associates it with its member(s). Members may have different responsibilities within the session (e.g. management or purely interaction with the service content and general session control). If a service session is the responsibility of an access session, the service session can remain active while that access session is active. When an access session is ended, related usage service sessions must be ended, suspended, or transferred to another access session or domain access session.

5.6.1 Provider Service Session (PSS)

It contains a central view of the service session, including all members and any additional provider information and logic necessary to execute service requests and maintain the session. Support of this session is the responsibility of the provider. A provider service session represents the service capabilities common to multiple members. Generally the provider service session holds information objects related to the management view of the service (e.g. accounting) or system related information of the service.

5.6.2 Usage Service Session (USS)

This is the member's (e.g. an end-user's) customized view of a service. It hides service complexity from the member and ensures that the member's preferences and environment are supported by the service. It hides the heterogeneity of each usage configuration from the provider service session. It will be further decomposed into domain usage parts.

5.6.3 Domain Usage Service Session (D_USS)

This object is an abstract object which represents generic information required to establish and support a service between two domains for an associated session member. Each domain usage service session is associated with a particular domain and a particular service session. As this object is an abstract object, it is not instantiable. The D_USS is specialized into four types of new information objects as described in the following sections.

5.6.3.1 User Domain Usage Service Session (UD_USS)

This is the functionality and information present in an end-user domain, e.g. a consumer domain, to support the usage service session and allow the end-user to interact with the service. In all cases, the UD_USS is associated with the usage party role. It is the responsibility of the end-user domain to deploy and manage this session. However, certain deployment and management responsibilities of resources in the UD_USS may be assigned to the provider domain by agreement.

5.6.3.2 Provider Domain Usage Service Session (PD_USS)

This is the functionality and information present in a provider domain (e.g. retailer, third party provider) to support the usage service session in a usage provider role. It hides the complexity and specifics of the other domain from the PSS and isolates party specific activity from the general activity of the PSS, which is common to all participants in the service session. The domain in the usage provider role is responsible for deploying and managing the provider domain usage service session.

5.6.4 Domain Usage Service Session Binding (D_USS_Binding)

It represents the dynamic information associated with binding two D_USSs. The D_USSs control this information. The information contained here is determined by the type of D_USS participating in the binding, and the session model(s).

5.7 Resource [Communication Session (CS)]

The Resource [Communication Session (CS)] represents a general, service view of stream connections and a network technology-independent view of the communication resources required to establish end to end connections. A Resource (communication session) can handle multiple connections which may be multi-point and multimedia.

A Resource (communication session) can arrange QoS, set-up, modify, and pull-down multiple connections.

The adoption of the "session" concept for controlling communication capability has the advantage of allowing services to instantiate dynamically, hold, resume and maintain a suitable configuration of communication resources that satisfies their needs.

A Resource (communication session) is controlled by one service session from the PSS or PD_USS. Only one service session may be associated with a Resource (communication session) at any one time.

6 Computational Model of the Session and Resource Level

6.1 Access Session Related Components

The access session related components support the access related sessions[4]. They support both the functionality and common session operations defined for the session concepts. A mapping between the session concepts and the service components is given in 6.4, "Relationship to Information Model.

Access sessions can be symmetric or asymmetric. The type of access session is determined by the reference point between the domains. This section starts by considering the components necessary to support an asymmetric access session. The user role is supported by the Provider Agent, while the provider role is supported by the Initial Agent and User Agent. A UAP is also considered, which support end-user needs.

6.1.1 User Application

The User APplication (UAP) is defined to model a variety of applications and programs in the domain. A UAP represents one or more of these applications and programs. A UAP can be used by human users, and/or other applications in the user domain. A UAP can be either or both an access session related and service session related component. The access session related UAP is defined below. The service session related UAP is defined in 6.2.

As an access session related component, the UAP enables a human user, or another application, to make use of the capabilities of a PA, through an appropriate (user) interface. An access session related UAP supports part of the domain access session. The UAP provides capabilities for:

– request authentication information from the user, required by the PA to set-up an access session with a UA;
– the user to request the creation of new service sessions;
– the user to request to join an existing service session;
– alerting the user to invitations, which arrive at the PA.

[4] Access Session (AS), Provider Domain Access Session (PD_AS), and User Domain Access Session (UD_AS).

An access session related UAP may also support the following optional capabilities, when they are also supported by the PA:

– allow the user to search for a provider, and register as a user of the provider's services;

– allow the user to search for services and identify providers providing those services.

Zero or more stream interfaces can be attached to a UAP. The stream interfaces can be bound to those in user systems and/or those in providers' domains.

A user domain contains one or more access session related UAPs. Any access session related UAP can request a PA to establish an access session. One or more UAPs interact with a PA to use its access session related capabilities within an access session.

A UAP instance may support only access session related capabilities or only service session related capabilities; or it may support both. Access session related UAPs may be specialized by a domain to interact with a specialized PA.

6.1.2 Provider Agent

The Provider Agent (PA) is a service independent component, defined as the user's end-point of an access session. The PA is supported in a domain, acting in an access user role. The PA supports a user accessing their UA and making use of services, through an access session. The PA supports the user domain access session, in conjunction with access session related UAPs, and other user domain infrastructure.

Capabilities supported by a PA:

– set-up a trusted relationship between the user and the provider (an access session), by interacting with an Initial Agent[5], and gaining a reference to a UA[6];

– within an access session:

  • convey requests (from a user to a UA) for creating new service sessions;

  • convey requests for joining existing service session;

  • receive invitations to join existing service sessions (from a UA) and alert the user[7];

  • anonymously make use of a provider's services;

  • deploy new components into the user's domain;

  • support access to terminal configuration information from a provider's domain;

  • register to receive invitations sent when no access session exists.

Operations supported by a PA are service independent.

---

[5] The PA may use a location service to find an interface reference for the IA, or some other means. The PA will provide the retailer name, and possibly other information to scope the search of the location service. The capability of the location service to return this interface reference is an important part in enabling access irrespective of location, which is one important feature of personal mobility. This assumes that the location service can indeed be contacted, irrespective of location. Also, it may imply that interworking between location services in different domains is required. How the location service gains an interface reference of an initial agent is undefined. It is likely that the location service has to interact with an object in the provider's domain in order to gain the reference. This interaction is not defined at present.

[6] This capability is an important element in support of personal mobility, as it allows a user to access a provider domain from various locations.

[7] Using an access session related UAP.

For each concurrent access session a user has with a provider, there is one PA instance in the user's domain. Each PA may be associated (through an access session) with the same UA, or separate UA instances. A single PA is only ever associated with one UA through an access session. (When no access sessions exist, a user domain can still support a PA. It can be used to initiate an access session, and may receive invitations if registered.)

6.1.3 Initial Agent

An Initial Agent (IA) is a user and service independent component that is the initial access point to a domain. An IA is supported by domains taking the provider role. An IA reference is returned to a PA when it wishes to contact the domain. The IA supports capabilities[8] to:

– authenticate the requesting domain and set up a trusted relationship between the domains (an access session) by interacting with the PA;

– establish an access session, but allowing the requesting domain to remain anonymous. The type of UA accessed in this way is an anonymous user agent.

Operations supported by an IA are service independent.

An IA supports requests from one PA at a time. The PA requests to contact the domain and is given a reference to an IA. When the PA has interacted with the IA to establish an access session with a UA, the reference to the IA will become invalid[9]. Subsequently, the IA may be contacted by another PA.

6.1.4 User Agent

A User Agent (UA) is a service-independent component that represents a user in the provider's domain. It is supported by a domain acting in the access provider role. It is the provider domain's end-point of an access session with a user. It supports the provider domain access session. It is accessible from the user's domain, regardless of the domain's location.

A UA supports capabilities to:

– support a trusted relationship between the user and the provider (an access session) by referencing the user's PA;

– within an access session:

  • act as a user's single contact point to control and manage (create/suspend/resume) the life cycle of service sessions and user service sessions;

  • create a new service session (by requesting that a service factory creates a USM and SSM);

  • join in an existing service session by creating a new user service session (via a service factory creating a USM);

  • resolve the service execution environment of the user, allowing them to use services from many different types of terminals. This requires resource configuration information of the user system (which includes terminals and their access points being used by or available for the user). Access to this information may be restricted by the user/PA;

  • provide access to a user's contract information with the provider;

---

[8] These capabilities are available irrespective of the location of the PA with which the IA interacts, and therefore are an important part of personal mobility support, i.e. allowing the user access irrespective of location.

[9] That is, the PA should not retain a reference to an IA after it has established an access session. An implementation may enforce that the reference to the IA is not usable once the access session has been established.

- resolve interaction problems between service usage requests.

The UA is defined as an abstract (i.e. non-instantiable) component type. Two instantiable subtypes are defined:

– Named User Agent (namedUA);

– Anonymous User Agent (anonUA).

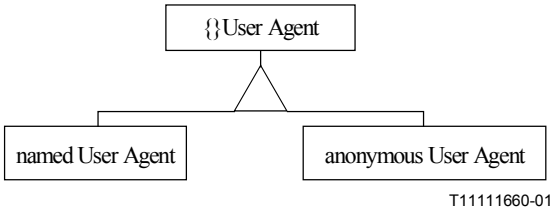


Figure 6-1 – Inheritance hierarchy for User Agent

Instances of the subtypes are created to represent different types of users. The subtypes of UA support all the capabilities which are defined for UA.

The namedUA is a UA specialized for a user that is an end-user or subscriber of the provider.

The anonUA is a UA specialized for a user that does not wish to disclose their identity to the provider.

Definitions for these subtypes are given in the sections below.

6.1.5 Named User Agent

A Named User Agent (namedUA) is a service independent component that represents a user in the provider's domain. The namedUA is a specialization of UA for a user that is an end-user or subscriber of the provider. It is the provider domain's end-point of an access session with a user. It is accessible from the user's domain, regardless of the domain's location.

The namedUA supports all of the capabilities which are defined for UA. In addition, it supports the following capabilities:

– within an access session:

- Act as a single contact point to control and manage (create/suspend/resume) the life-cycle of service sessions and user service sessions, taking into account restrictions posed by subscribers and the user;
- Suspend/resume existing user service sessions and service sessions. This includes support for session mobility;
- Manage the user's preferences (choices or constraints) on service access and service execution (This is supported by starting a provider specific service session.);
- Resolve the service execution environment for the user, allowing them to use services from many different types of terminals. This requires resource configuration information of the user system, (which includes terminals and their access points being used by or available for the user. Access to this information may be restricted by the user/PA.) This includes support for personal mobility;
- Register user at a terminal to receive invitations. This includes support for personal mobility;
- Allow the user to define user private/public policies. (This is supported by starting a provider specific service session.);

- Negotiate the session models and feature sets supported by a service session, in order for it to interact with a UAP in the user's domain.

– accept invitations from users to join a service session;

– deliver invitations to a terminal, previously registered by the user with the namedUA. No access session would be required to allow this delivery of invitations.

The namedUA may support the following optional capabilities:

– perform actions on the users behalf, when the user is not in an access session with the namedUA;

– initiate an access session with a PA;

– support additional authentication of the user. This may be tailored to the user, and usage context.

Operations supported by a namedUA are service independent.

A namedUA may support one or more access sessions concurrently[10]. Each access session is with a single, distinct PA.

6.1.6 Anonymous User Agent

An Anonymous User Agent (anonUA) is a service independent component that represents a user in the provider's domain. The anonUA is a specialization of UA for users that do not wish to disclose their identity to the provider. It is the provider domain's end-point of an access session with the anonymous user.

The anonUA supports all of the capabilities which are defined for UA. In addition, it supports the following capabilities:

– Support a trusted relationship between the user and the provider (an access session) by referencing the user's PA. The provider does not know the identity of the user. ('Trust' is not guaranteed by identifying the user, as for the namedUA, but may be ensured by, for example, pre-payment.);

– within an access session:

- Suspend/resume existing user service sessions and service sessions within an access session. (Suspended sessions cannot be resumed in a different access session.[11]);
- Manage the user's preferences (choices or constraints) on service access and service execution. (These would have to be determined during the access session, and could not be re-used in a separate access session.);
- Provide access to a user's contract information with the provider. (This contract would be defined at the start of the access session and terminated at the end of the access session.);
- Define user private /public policies. (This may be supported by starting a provider specific service session. This information would only be maintained during this access session.);
- Allow the anonymous user to register as a user of the provider (i.e. set-up a contract with the provider for longer than a single access session);
- Negotiate the session models and feature sets supported by a service session, in order for it to interact with a UAP in the user's domain.

---

[10] NamedUAs must be able to support one access session, and may optionally be able to multiple concurrent access sessions. NamedUAs continue to exist when there is no access session.

[11] It is assumed suspended sessions are ended by the provider if the access session is ended.

The anonUA provides no support for personal or session mobility.

6.2 Service Session Related Components

The service session related components defined in this section follow the session concepts in [6]. Service sessions can be supported over multiple domains. The service related sessions[12] are supported by these components.

The interactions between the domains depend on the role the domain takes in the service session. A domain acting in a usage provider role will support a Service Session Manager. It will also support User Service Session Managers (USMs) for each domain acting in a usage party role. The party domain supports a service session related UAP to interact with the USM.

A domain may be perceived as a party domain by the provider domain, while it is actually composing this service session with its own.

Session models define how service session components in each domain can interact in a generic manner. These session models allow components, which have been designed and implemented separately, to interact to support the service session.

The session model allows service session components to make requests about: ending and suspending the session, the parties involved, set up and modification of stream bindings between parties, for example.

Service session related components may support one or more of a variety of session models. These session models may be defined by a variety of organizations. Each session may support a number of session models, or may only support a single model. Services may decide not to support the proposed Session Model. This is acceptable because the access part includes the possibility to negotiate alternative usage interfaces.

6.2.1 User Application

The User APplication (UAP) is defined to model a variety of applications and programs in the user domain. A UAP represents one or more of these applications and programs in the computational model. A UAP can be used by human users, and/or other applications in the user domain. A UAP can be either or both an access session related and service session related component. The access session related UAP is defined in 6.1.1. The service session related UAP is defined below.

As a service session related component, the UAP enables a user to make use of the capabilities of a service session, through an appropriate user interface. It acts as an end-point of a service session, by supporting the User Domain Usage Service Session (UD_USS). The capabilities provided by particular UAPs are specific to the UAP, and any service session it is part of. UAPs may provide some of the following generic capabilities to the user, such as:

- starting/ending the session;
- inviting other users to join the session;
- joining an existing service session;
- adding/removing/modifying stream bindings and the users' participation in them;
- establishment of control session relationships and other changes in the service session;
- suspending the user's participation in the session, or the whole session;
- resuming the user's participation in the session, or the whole session.

---

[12] Service Session (SS), Provider Service Session (PSS), Usage Service Session (USS), Provider Domain Usage Service Session (PD_USS), and User Domain Usage Service Session (UD_USS).

Zero[13] or more stream interfaces can be attached to a UAP. The stream interfaces can be bound to those in other user systems and/or those in the provider domain.

The user's domain contains one or more service session related UAPs. One service session related UAP can be involved in one or more service sessions. For each service session the UAP is involved in, it interacts with a user service session manager, or a service session manager[14].

The UAP may also support a particular session model, such as the proposed Session Model. The USM/SSM uses these interfaces to share information with the UAP on parties and service resources in the service session.

A UAP instance may support: only access session related capabilities; only service session related capabilities; or it may support both. Service session related UAPs will be specialized to the service session(s) they interact with. In order to use a service session, a UAP specialized to the service type must be present in the user's domain (or be deployed, e.g. by downloading) before the session is used.

6.2.2 Service Factory

A Service Factory (SF) is a service-specific component that creates the service session components for a service type.

A request to create a service session of a particular service type will result in the creation of one or more service related component instances[15]. The SF will create and initialize the instances according to rules imposed by their implementation. The SF will return to the client one or more interface references to these components. (The SF is used to create instances of all the service session related components defined in this document: USM and SSM.)

Requests are typically made by UAs. Other clients may also be able to request the creation of a service session. The client must have an interface reference to the SF and issue an appropriate request. A SF which supports more than one service type would typically provide separate interfaces for each service type.

A SF supports capabilities to:

– create service related components for one or more service types upon request. (This includes choosing the session models supported by the service session, although this may be fixed by the service type.)

SF may support optional capabilities to:

– create a service related component (typically USM) to be used in conjunction with other service related components (typically a SSM & USMs) created by a different factory instance;

– continue to manage the created components. It may provide a list of sessions managed by it, and may 'clean up' some sessions if requested;

– may include mechanisms to schedule the activation of a session at a specific date and time. (This mechanism includes resource reservation.);

– support suspension/resumption of a service session.

---

[13] Some services don't require stream interfaces on the UAP.

[14] Services which only support a single user in a service session can provide only an SSM (no USM), and allow the UAP to interact directly with the SSM. These services will be restricted to only ever having a single user in a service session, and should not be able to invite other users to join the session, as no USM for the invited user could be made available.

[15] Typically, the USM and SSM.

The SF assembles the resources necessary for the existence of a component it creates. Therefore, the SF represents a scope of resource allocation, which is the set of resources available to the SF. A SF may support an interface that enables clients to constrain the scope.

6.2.3 Service Session Manager

The Service Session Manager (SSM) is a component which comprises the service-specific and generic session control segments of the Provider Service Session (PSS). An SSM supports service capabilities that are shared among members (parties, service resources, etc.) in a service session. Information related to a particular member of the service session are encapsulated in Member Usage Service Session Managers (MUSMs). SSMs support (some or all of) the following capabilities:

- keep track and control the various resources shared by multiple users in a service session. This can be done just by having references to other objects (like a CSM) which really maintain the context of usage for a specific kind of resources;
- hold the state of the service session and support suspension/resumption of the service session;
- support adding/inviting/removing users to/from the service session by interacting with the corresponding UAs;
- support adding/removing/modifying stream bindings and the users' participation in them;
- support the negotiating capabilities among the users interacting with the USMs. SSM will serve as a control center of consensus building (such as voting procedures);
- support management capabilities associated with the service session (e.g. accounting).

Zero[16] or more stream interfaces can be attached to an SSM. The stream interfaces can be bound to other stream interfaces in this or other domains.

An SSM is created by a SF, one per request for the corresponding service type. It is deleted when all the users leave the service session, or when quit by a user or SF. The life span of an SSM is the same as the corresponding provider service session.

6.2.4 Member Usage Service Session Manager

The Member Usage Service Session Manager (MUSM) is an abstract component, which comprises the service-specific and generic session control segments of the Domain Usage Service Session (D_USS) that interact with the PSS. It is specialized according to the role of session member supported by the D_USS:

- User Service Session Manager (USM) represents the UD_USS;

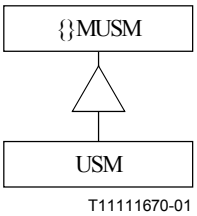


Figure 6-2 – Inheritance hierarchy for Member Usage Service Session Manager

[16] Typically stream interfaces are offered by Service Support Components (SSCs) associated with an SSM.

The MUSM represents and holds the context of a member (party, service resource, or provider) in a service session. Its main characteristics are the following:

– It contains the information and service capabilities which are local to the member. If an operation involves activities that are purely local to the member, the MUSM controls and manages the activities by itself. If not, the MUSM interacts with the SSM to perform the operation. The SSM may interact with the MUSM in response to operations from other members (or due to service logic) that affect this member. Such interactions are dependent on the role of the member;

– It keeps track of and controls the exclusive (non-shared) resources used by the member in a service session. This can be done just by having references to other objects (e.g. a communications session manager) which really maintain the context of a member for a specific kind of resource;

– It may be configured to preferences of the member. This may be done during initialization, and dynamically during the session.

As the MUSM is an abstract service component, no instances are created. Instances of the appropriate specialized component are created to represent specific session members.

6.2.5 User Service Session Manager

The User Service Session Manager (USM) is a component which comprises the service-specific and generic session control segments of the Provider Domain User Service Session (PD_USS). It is a specialization of the MUSM which represents and holds the context of a party[17], or resource in a service session. It has the same characteristics as the MUSM (with member replaced by party or service resource as applicable):

– It holds the state of the PD_USS and supports suspension/resumption of the party's participation in the service session;

– It supports the different roles of the party in the service. The role of a party will be service dependent (e.g. chairman in a conference).

Zero[18] or more stream interfaces can be attached to a USM. The stream interfaces can be bound to other stream interfaces in this or other domains.

A USM is created by the SF, one per request for the corresponding service type (per PD_USS). It is deleted when the party leaves the service session. The life span of a USM is the same as the corresponding PD_USS.

6.3 Resource (Communication Session) Related Components

Note that services not using stream bindings will not have Resources (communication sessions) and will not need these components. However, as the components are service independent, it is likely that a CSM or TCSM will be present in most domains.

---

[17] A party can be either an end-user, or a service session acting in a usage party role.

[18] Typically stream interfaces are offered by Service Support Components (SSCs) associated with a USM.

6.3.1 Communication Session Manager

The Communication Session Manager (CSM) is a service-independent component which manages application-level, end-to-end bindings between stream interfaces (stream flow connections). A stream flow is an abstraction of a connection. The CSM provides interfaces to allow USM/SSMs to set-up, modify, and remove stream flows. The CSM decomposes the requested connection into two parts, the nodal part and the transport part. It requests TCSM to take care of the nodal part and requests other connection management objects to take care of the transport part.

The CSM provides the following capabilities:

– creation and control of stream flow connections (SFCs), end to end.

6.3.2 Terminal Communication Session Manager

The Terminal Communication Session Manager (TCSM) is a service-independent component that manages Terminal Flow Connections (TFCs) (intra-nodal flow connections) within the user's domain. The TCSM provides an interface to the CSM, to allow the CSM to request the TCSM to set-up, modify, and remove connections in the user's domain.

The TCSM provides the following capabilities:

– creation and control of flow connections (TFCs) within the user domain.

6.4 Relationship to Information Model

Tables 6-1 and 6-2 provide a mapping between session concepts and objects in the information model, and components in the computational model.

Session concepts, which are mapped to a component, means that the component supports the functionality and state of the session, and controls the service resources which are part of the session. If a session concept is mapped to several components, then each of the components support part of the functionality and state, and control some of the service resources of the session.

Information objects which map to a component mean that the information in the information object is contained within the component, and that the component may provide access to that information to other components/objects.

Table 6-1 – Mapping between access session related components

Session Concept/Information Objects	Components
Access Session (AS) with User-Provider Roles	PA and UA
User Domain Access Session (UD_AS)	PA
Provider Domain Access Session (PD_AS)	UA
User Profile with User-Provider Roles	UA
Contract with User-Provider Roles	PA and UA

Table 6-2 – Mapping between service session related components

Session Concept/Information Objects	Components
Service Session (SS)	Service session related UAP, USM, and SSM
User Service Session (USS)	Service session related UAP, and USM
User Domain Usage Service Session (UD_USS)	Service session related UAP
Provider Domain Usage Service Session (PD_USS)	USM
Provider Service Session (PSS)	SSM

6.5 Examples

Example scenarios are described in the following sections to illustrate how the components can interact in the following cases:

– contacting a provider;

– logging in to a provider as a known user;

– starting a new service session;

– inviting a user to join an existing service session;

– joining an existing service session;

– creating a stream binding in an existing service session.

Note that the scenarios are examples and that they assume all the operations are successfully completed (no error, no fault, and no rejection) for simplicity.

6.5.1 Contact a Provider

This example shows the user A making contact with a provider (see Figure 6-3). This scenario supports user mobility by allowing the user to contact a specific provider from any terminal.

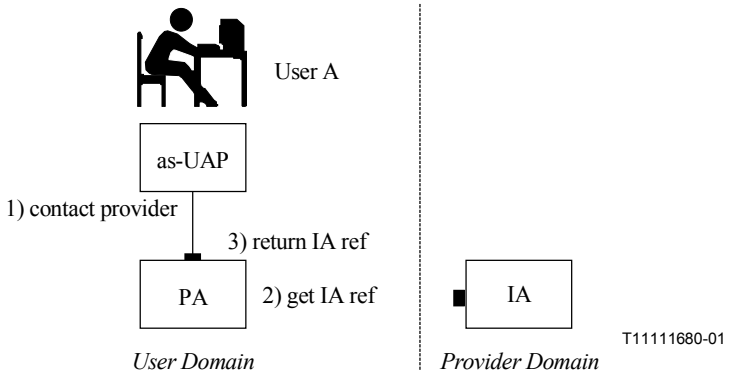


Figure 6-3 – Contacting a provider

Preconditions:

A PA must be present in the user's domain.

Scenario:

1) User starts an access session related UAP. He provides the retailer name he wishes to contact. UAP requests the PA to contact the provider, giving the retailer name.
2) PA gains a reference to an interface of an initial agent of the provider[19].
3) PA returns success to UAP.

Post-conditions:

The PA has an interface reference to the IA. The user has not setup an access session between the PA and IA. The IA does not support use of a user's services, only operations to set up an access session as a known user (see 6.5.2) or an anonymous user.

The IA has no knowledge of any interfaces on the PA or in the user's domain.

It is possible for the provider to download a provider specific PA to the user's domain, once an interface reference to the InitialAgent has been gained by the PA. This helps to support user mobility. No scenario describing how this is achieved is defined at present.

6.5.2 Login to a Provider as a Known User

This example shows the user A establishing an access session with their named user agent of the provider (see Figure 6-4). The user wishes to make use of the provider's services which the user has previously subscribed to (see 6.5.3). This scenario supports user mobility by allowing the user to establish an access session with a provider from any terminal.

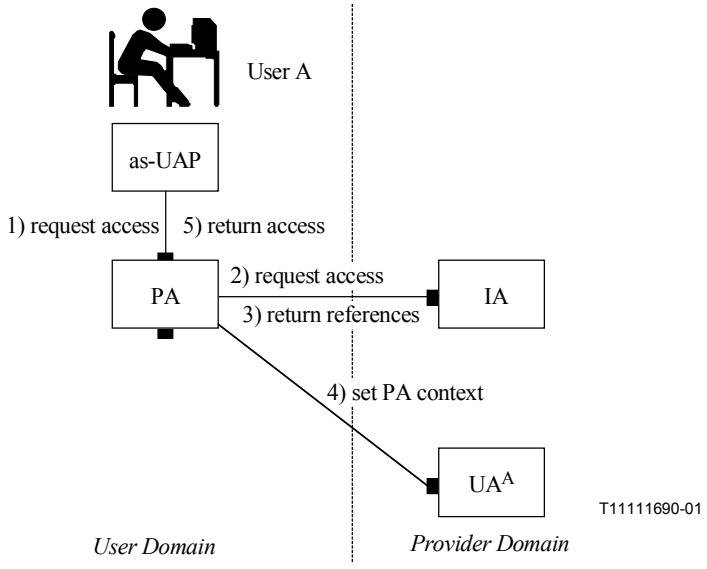


Figure 6-4 – Login to a provider as a known user

[19] The PA may use a location service to find an interface reference for the IA, or some other means.

Preconditions:

The user has contacted the provider (as in 6.5.1), and the PA has an interface reference to an initial agent of the provider.

Scenario:

1) User A uses an access session related UAP to login to the provider, as a known user[20]. The user has then requests the PA to login to the provider, as a known user. The UAP supplies the security information to the PA.
2) PA requests that an access session be set up with the namedUA of the user. PA provides the username of the user to the IA. The PA has passed the user security information to the security services supported by the DPE. The security services interact with the provider domain in order to authenticate the user[21].
3) IA has already authenticated the user through the DPE security services, and an access session has been established. It returns the interface reference of the user's UA.
4) PA sends information about the user domain to the UA. This information is termed the PA context, which includes references to interfaces on the PA, and possibly terminal information.
5) PA returns success to UAP.

Post-conditions:

User has setup an access session between the PA and namedUA. The namedUA is personalized to the user, and has knowledge of interfaces of the PA.

Any interface references of the IA held by the PA will be invalid.

Note that in the sequence of events, personal mobility is allowed due to the following capabilities:

a) The capability of the naming service to be contacted irrespective of location and to return a reference to the IA;
b) The IA to establish a trusted relationship with the user, that is independent of the physical location of that user;
c) The IA to return a reference of the user's own named UA to the (PA acting on behalf of the) user.

It is possible that once an access session has been established, the provider may download a provider specific PA to the user's domain. This helps to support user mobility. No scenario describing how this would be achieved is defined at present.

6.5.3 Starting a new Service Session

This example shows a user starting a new service session (in this example, a videoconference service, but the interactions would be the same for all types of service). The user is assumed to be in an access session with the provider and to have a valid subscription to the service (the service type is videoConference234). The service session related UAP is assumed to be present on the user's terminal. (see Figure 6-5).

---

[20] UAP may ask the user for their username and other security related information, e.g. password.

[21] If security services were not supported by the DPE, then the PA would first have to send authentication information to the IA.

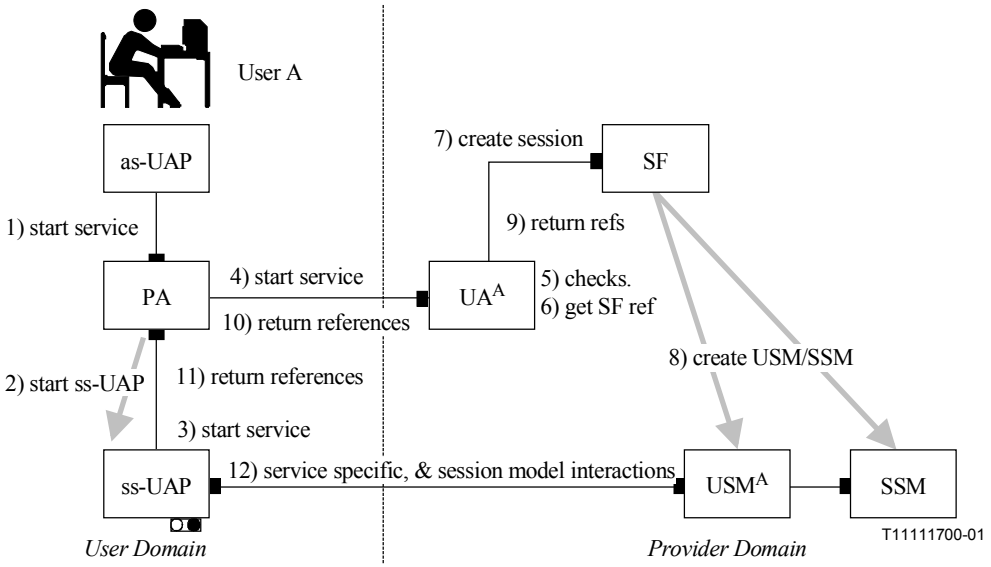


Figure 6-5 – Starting a new service session

Preconditions:

An access session exists between the PA (user A) and UA (in provider domain). An access session related UAP shows the user the services which he can start.

Scenario:

1) The access session related UAP requests a list of services from the PA, which the user has been subscribed to. The PA makes the same request to the UA, which returns the list. The UAP displays the list to the user. The user selects a service to start[22]. The UAP requests PA to start the service.

2) The PA starts the service session related UAP[23], associated with this service session, and informs it of the service type that it should start (videoConference234).

3) The service session related UAP requests a new service session of service type videoConference234, from the PA. (The UAP may pass information about itself to the PA, including session models and feature sets supported and references to its operational and stream interfaces.)

4) PA requests to start a new service session of the service type (videoConference234), to (user A's) UA. (It may also pass the information about the UAP.)

5) UA may perform some actions, which are not prescribed here, before continuing. For example, the UA checks the new session request against the user's subscription profile[24], to verify that the user has subscribed to this service and that it can be used with the terminal configuration of the user. Other decisions may also be taken. UA raises an exception to the PA, if the UA declines to start the service session.

---

[22] The preceding interactions are not shown in the figure.

[23] If the service session related UAP is not available in the user's domain, PA may attempt to download the UAP and continue.

[24] The user's subscription profile may define preferences and constraints on the invocation of a service. These preferences/constraints may be dependent on the user's current location. This provides support for personal mobility.

6) UA gets a reference to a service factory which can create service session components for the service type (videoConference234).[25]

7) UA requests that a new session of the service type (videoConference234) be created by the Service factory.

8) Service factory creates an SSM and a USM[26] and initializes them.

9) Service factory returns interface references of the USM and the SSM to the UA.

10) UA returns references of the USM and SSM to the PA.

11) PA returns references of the USM and SSM to the service session related UAP.

12) The service session related UAP and USM (and SSM) can interact using service specific interfaces or interfaces defined by session models, including the proposed session model. Some interactions between these components may be necessary before the user can use the service.

At this point User A is the only user involved in the service session. Some services may be single user only services, or may be used by a single user. As this is an example of a video conference service, user A probably wants to invite some more users to join in the session.

6.5.4 Inviting a User to Join an Existing Service Session

This example shows user A inviting another user (B) to join in the service session (see Figure 6-6). The example ends when the invitation has been delivered to user B. The example of user B actually joining the session is given in 6.5.5.

This example assumes that the invited user B is a named user, and is represented in the provider by a named user agent. Anonymous users, represented by an anonymous user agent, cannot be invited to join a service session because it is not possible to locate the specific user, as anonymous user agents do not publish the identity of the user (and may not even know the user's identity).

This scenario supports user mobility by allowing a user to be invited to join a session, irrespective of their location. (This does not mean that they will automatically be able to join the session.)

---

[25] The UA may use a location service to find an appropriate service factory, or some other means. The UA may also provide other information to scope the search for the service factory, such as terminal configuration information. Other means include: the subscription information could potentially contain an interface reference to the service factory to use.

[26] The Service Factory creates the computational objects which comprise the service session. These may include the USM and SSM. Other computational objects are also possible.

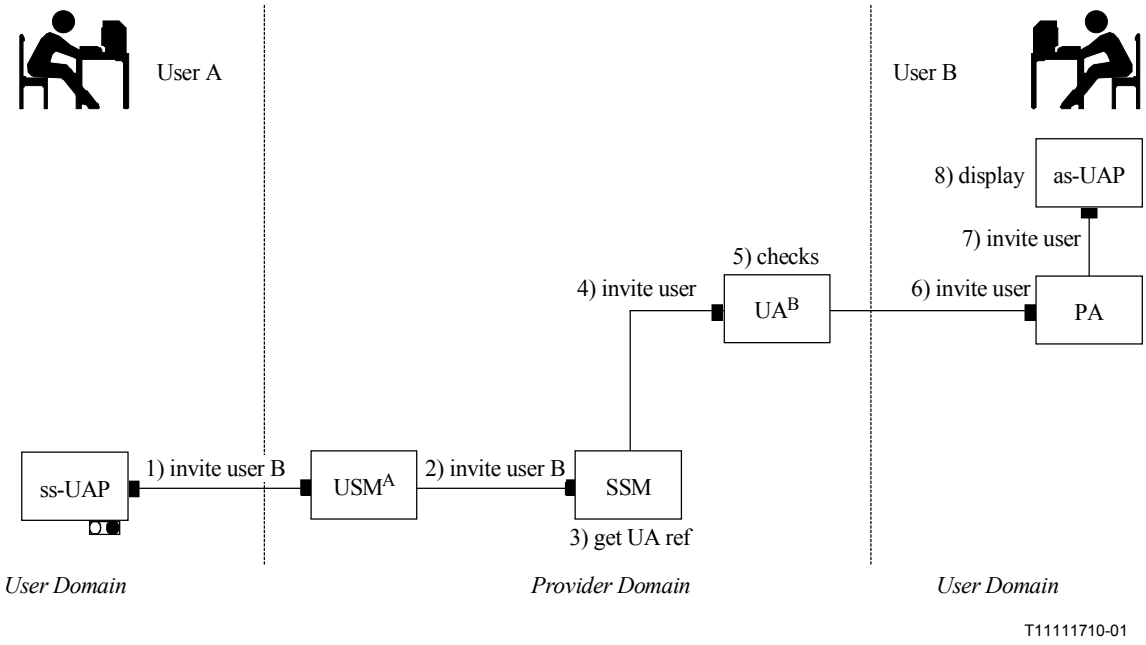


Figure 6-6 – Inviting a user to join an existing service session

Preconditions:

An access session exists between the PA and UA of the user sending the invitation (user A). It is NOT necessary for an access session to exist between a PA and the UA of the user receiving the invitation (user B), but for this example assume that an access session does exist for user B.

User A is using a service session related UAP and has a service session established with a USM and SSM. User A wishes to invite user B to join this service session. User A is 'active' in the service session, i.e. they have not suspended their participation.

Scenario:

1) User A uses UAP to invite another user (invitee) to join a session. (User A supplies the user name of the invitee, or a user defined alias which can be resolved by the inviter's UA.) The UAP requests the USM to invite user B to join the session.
2) USM requests the SSM to invite a user to join the session. (Both the USM and SSM may check that User A is allowed to invite User B. These checks are not defined here.)
3) SSM gets a reference to an invitation interface of user B's UA[27].
4) SSM sends an invitation using the invitation interface of the user B's UA.
5) Invitee's UA may perform some actions, which are not prescribed here, before continuing. For example, the UA may check the user profile within the UA for a policy on invitations. The policy will then determine the UA actions and interactions with other objects. UA may raise an exception to the SSM, if the UA declines to deliver the invitation.
6) In this example an access session exists between user B's UA, and the PA on User B's terminal. The invitation is delivered to the PA, by using an invitation interface on the PA.
7) PA sends the invitation to the access session related UAP.
8) The UAP displays the invitation to user B.

[27] The UA may use a location service for locating user B's UA, or some other means.

The invitation to join the service session has been delivered to user B's UA, PA, and is displayed by the UAP. The invitation contains sufficient information for the UA to locate the service session, and allow the user to join it (as described in 6.5.5). Only some of this information will be passed to the PA and UAP.

In the example above an access session already existed between user B's PA and UA. If user B is NOT currently in an access session with the UA, then there are several alternatives as to what happens. It is not currently defined which of these alternatives must be supported as a mandatory capability and which are optional. The alternatives are:

- UA stores the invitation until the invited user establishes an access session. When he does establish an access session, the invitation is delivered as above;
- UA delivers the invitation to a registered terminal. (The terminal would have been selected by the user to receive invitations when no access session was present)[28];
- UA returns the address of a registered terminal to the SSM;
- UA forwards the invitation to another UA. (This UA would have been selected by the user to receive invitations when no access session is present. The UA may be in a different provider's domain.);
- UA returns the address of another UA;
- UA starts a service session of a specified type. (The invitation may be sent to the service session, as part of its configuration, or later.).

6.5.5 Joining an Existing Service Session

This example shows a user B joining an existing session, after receiving an invitation to join the session (see Figure 6-7).

User B is assumed to be in an access session with the provider and to have a valid subscription to the service (the service type is videoConference234). The service session related UAP is assumed to be present on user B's terminal.

User B can join this session from any terminal, from which he has established an access session. When the access session is established, the PA requests a list of the invitations received by the UA. The PA can then request to join any of the sessions. However, in this example we assume the invitations have been delivered to User B's PA and as-UAP as described in 6.5.4.

[28] This case is required to support personal mobility.

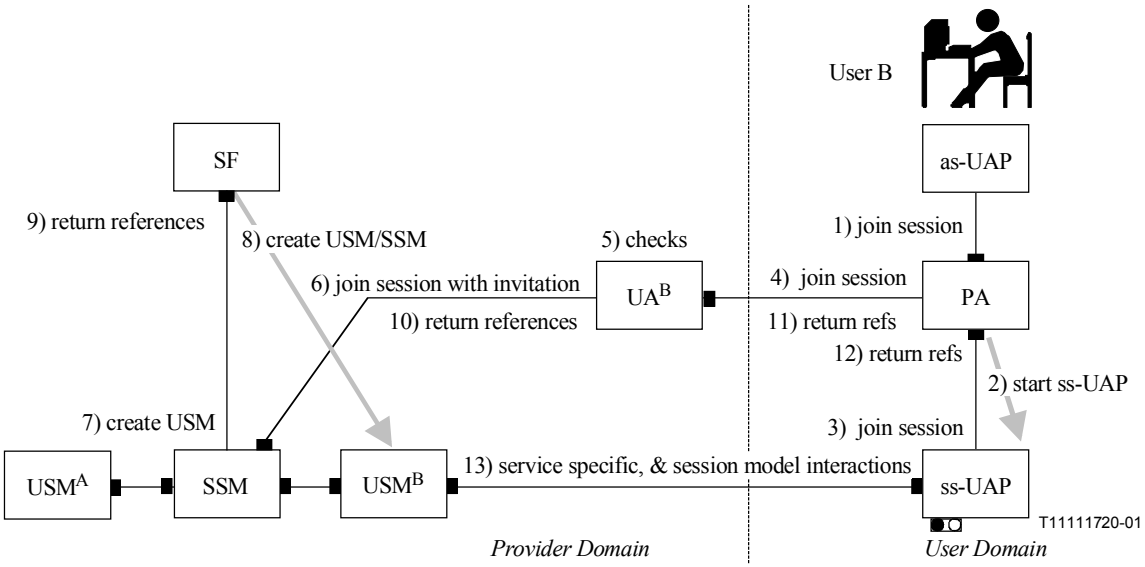


Figure 6-7 – Joining an existing service session

Preconditions:

An access session exists between the PA (user B) and UA (in provider domain). User B's UA and PA have received the invitation to join the service session, and an access session related UAP shows the user the invitation which he has received.

Scenario:

1) The access session related UAP displays a list of invitations to join service sessions. The user selects an invitation to join the service session. The UAP requests PA to join the service session, giving the invitation id.

2) The PA starts a service session related UAP[29], associated with this type of service session, and informs it of the invitation id that it should request to join.

3) The service session related UAP requests to join the service session, giving the invitation id, from the PA. (The UAP may pass information about itself to the PA, including session models supported and references to its operational and stream interfaces.)

4) PA requests to join the service session, giving the invitation id, to (user B's) UA. (It may also pass the information about the UAP.)

5) UA may perform some actions, which are not prescribed here, before continuing. For example, the UA may check the invitation id against the user's current invitations, as well as the user's subscription profile[30] to verify that they are subscribed to this service, and that it can be used with the current terminal configuration, etc. UA can decline the user to join the session.

[29] If the service session related UAP is not available in the user's domain, PA may attempt to download the UAP.

[30] The user's subscription profile may define preferences and constraints on the invocation of a service. These may be dependent on the user's current location. This provides support for personal mobility.

6) UA gets a reference to the SSM[31] and requests to join the session. (It may pass some information in the invitation to confirm that the SSM invited this user to join the session.)

7) SSM requests its service factory to create a USM for user B.

8) Service factory creates a USM and initializes it.

9) Service factory returns interface references of the USM to the SSM.

10) SSM returns references of the USM and itself to the UA.

11) UA returns references of the USM and the SSM to the PA.

12) PA returns references of the USM and the SSM to the service session related UAP.

13) The service session related UAP and USM (and SSM) can interact using service specific interfaces or interfaces defined by session models, including the proposed session model. Some interactions between these components may be necessary before the user can use the service.

At this point both user A and user B are involved in the service session. As this is an example of a video conference service, either user may invite other users to join the session. There may be some service specific policy to decide whether a particular user in the session is allowed to invite other users to join.

6.5.6 Request and Establishment of a Stream Binding

This example shows the set-up of a stream binding between UAPs in consumer 1 and 2's domains (see Figure 6-8). Consumer 3 requests the establishment of a stream binding in which consumer 1 and consumer 2 shall participate, but doesn't himself participate in the stream binding[32].

---

[31] The invitation may contain a reference to an interface on the SSM to use to join the session. Or the UA may use information in the invitation along with a location service to find the SSM, or some other means. The UA may also provide other information to the SSM, such as terminal configuration information, and application information.

[32] This example is chosen to illustrate the separation of service session and communication session. Of course other examples are possible, e.g. one (more 'POTS-like') where consumer 1 acts as the initiator of the service session and also as the initiator of the communication session. The separations of access and usage and of service session and communication session still have meaning and add value (e.g. support mobility aspects).

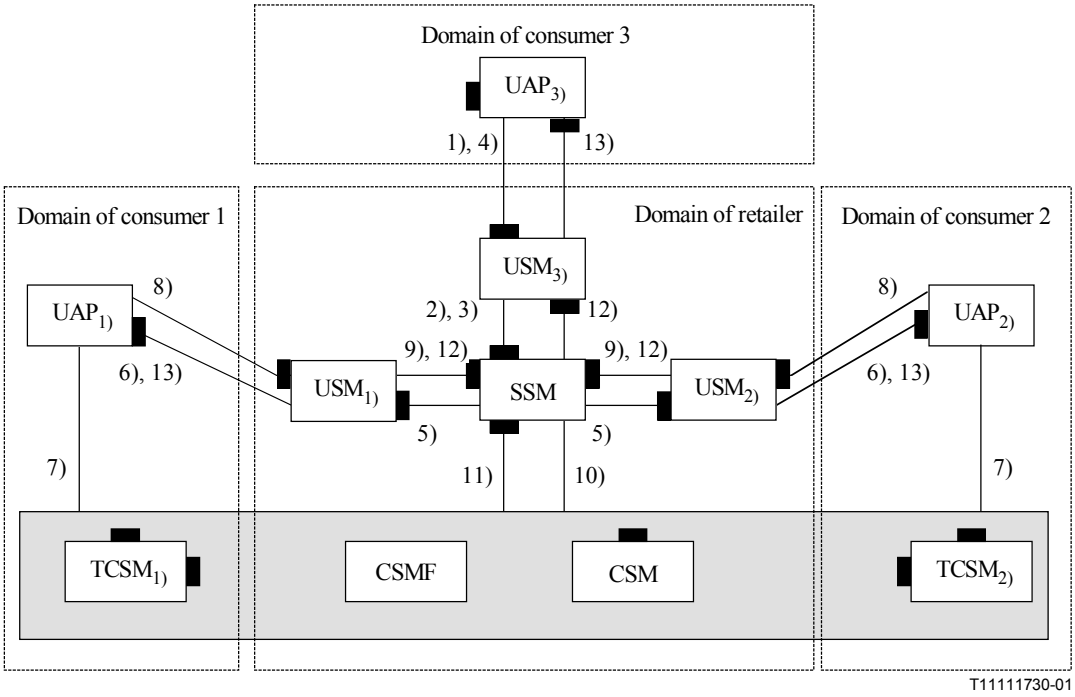


Figure 6-8 – Stream binding request and setup [3 parties in the service session, Resource (communication session) involving 2 of them]

Precondition:

The 3 consumers have already been invited to the service session and are now parties in the service session.

Post-condition:

A stream binding is established between the two participants, party1 and party2, and party3 has a reference to the stream binding in order to control it further.

Scenario:

The scenario shows a successful setup, but at certain points it is made clear that different decisions could have been made.

1) UAP3 requests the setup of a stream binding with party1 and party2 as the participants. USM3 may optionally make the necessary checks to make sure consumer 3 is allowed to setup the requested stream binding.

2) USM3 forwards the request to the service session's SSM.

   Optionally, SSM may check for permission to set up this stream binding; if necessary it may negotiate for permission with the other session members in the session. [This will then involve the voting feature set (not shown)].

3) If permission is obtained, SSM returns a stream binding identifier, as well as a request identifier for later confirmations to USM3.

4) USM returns a stream binding identifier, as well as a request identifier, for later confirmations to UAP3.

   (The following can be done in parallel to '1' and '2', hereafter called 'i')

5) SSM requests USMi to join the stream binding.

   USMi may optionally make decisions, e.g. about non-participation on behalf of consumer i.

6) USMi forwards the request to UAPi.

7) UAPi starts an application setup scenario to get the NFEP(pool)s related to the stream interface user by consumer i in this service. This may have been done already[33], or it must be done now in order for the consumer to participate in this stream binding.

8) UAPi accepts and returns this acceptance to USMi, together with a description of consumer i's terms of participation in the stream binding, as well as a stream interface descriptor.

9) USMi forwards this acceptance and the associated information to SSM.

Depending on the answers from the participants (and specific logic for this service), the SSM may choose to give up establishing the stream binding, or the stream binding request will result in a request for communications (this is what is shown).

10) SSM requests to setup a Resource (communication session) (if not already existing), and then requests the stream flows associated with the stream binding to be set up.

11) Final notifications back to SSM.

12) Final notifications back to USMs for consumer 1, 2 and 3 [or just some of them, depending on their reply in step 9 (for each consumer i)].

13) Final notifications back to consumers 1, 2 and 3 (or just some of them, depending on the reply from UAP in step 9, and/or depending on the behaviour of USMi[34].)

7 Overview of Ret Specification

This clause introduces the specification, gives the overall functionality and scope of the reference point involved in this Supplement, and briefly defines this reference point.

Clause 8 describes how a consumer accesses a retailer to make use of services they make accessible. It addresses the establishment, and use of a secure association between the domains, termed an Access Session (defined and described in more detail in clause 8.). Within the access session, it addresses the control of services, and service sessions, and the subscription management. It consists of a set of operational interfaces, offered by the consumer and by the retailer business roles. Interfaces are first defined informally using plain text and diagrams, then by means of semi-formal IDL specifications; behaviour is described in plain text. An interface dedicated to subscription is described in detail as well. Complete specifications of IDL interfaces are given in clause 8.

Clause 9 contains the complete specifications of IDL interfaces for Ret-RP including the interfaces for the subscription part.

This Supplement consists of non-formal specifications, in terms of plain text and diagrams, and of semi-formal specifications, using the Interface Definition Language.

The purpose of this Supplement is to provide specifications ready to be used for interoperable multi-vendor implementation of the computational interfaces required between the domains described in this Supplement: the Retailer Domain, and the Consumer Domain.

---

[33] Depending on the service type and the terminal capabilities, there might be several cases: If the application is the only one ever using streams, the nodal (terminal) part of the stream binding may be hardwired. It might also be the case that, when receiving the invitation, consumer 1 already knows (or finds it likely) that he will be asked to participate in a stream binding, and starts to prepare the terminal internal actions needed to get hold of a stream interface (e.g. ask another application to release a stream interface or kill some applications in order to increase performance). This shows specialization of behaviour of the UAP.

[34] It is possible that the USM gets the notification, but does not forward it to the UAP; this is similar to what is explained in step 5), where USM takes decisions (e.g. 'screens') on behalf of the user/UAP.

7.1 Overall functionality and scope of the reference points

The definition of a reference point in SPFEE is that it defines the interactions between stakeholders (by means of interfaces they provide one to another). In this Supplement:

– Retailer reference point (Ret-RP): between the Consumer and the Retailer.

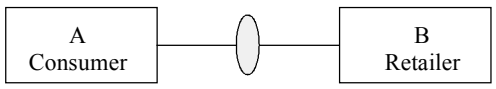
Note that the End-User role is generalized in this Supplement as Consumer role. The stakeholder role of consumer models two stakeholders: the Subscriber and the End-User. The Subscriber is the entity that has a business relationship with the Retailer, whereas the End-User is the person that actually makes use of the capabilities provided by the Service Provider through the Retailer.

The Ret-RP supports a consumer accessing a retailer to make use of services they make available on behalf of one or more Service Providers. It addresses the establishment, and use of a secure association between the domains, termed an Access Session. Within the access session, it addresses the control of the life-cycle of the usage of the Service Provider services. It corresponds to the functionality, interfaces and objects related to the access session. It defines interfaces to support use of the following functionality:

– initiation of dialogue between the consumer (subscriber and end-user) and retailer domains;
– identification of the domains to each other (either domain can remain anonymous dependent on the interaction requested);
– establishment of a secure association between the domains, an access session;
– set up of the default context for the control and management of usage functionality;
– discovery of service[35] offerings;
– listings of access sessions, service sessions and subscribed services;
– initiation of usage between the End-User and Service Provider domains;
– control and management of sessions (e.g. stop, suspend, resume, join, notify changes, etc.).

The following principles are used in this Supplement:

– Personal and session mobility, providing the description of how to transfer and manage personal environments between end-user access points inside a session;
– Management, providing the mechanisms to manage both administrative information (e.g. subscribers) and FCAPS (e.g. fault management for a service).



Use of the Ret-RP with respect to the TINA business roles

T11111740-01

[35] These services can be primary [e.g. Video on Demand (VoD)], ancillary to the primary (e.g. configuration management for VoD) or administrative (e.g. subscriber management for VoD).

7.1.1 Ret-RP business role life-cycle

The Ret-RP supports the whole lifecycle of the relationship between consumer and retailer, which is described as Subscriber and End-user life-cycle.

The Subscriber lifecycle describes the processes by which a Subscriber establishes a relationship with a retailer, and modifies or terminates the relationship. The relationship includes subscription, customization, and the association between Subscriber and End-user.

The End-user lifecycle describes the process by which end-users can access and use services. This includes end-user system setup, retailer contact, and service customization.

7.2 Main assumptions

Two main assumptions in this Supplement are:

– a pervasive, interoperable, DPE is assumed, providing security services;

– the existence of a naming addressing and resolution framework are assumed.

7.3 Definition of Ret Reference Point

The Ret reference point definition is a semi-formal specification of the business relationship between the consumer and retailer business roles. Conforming to the SPFEE Business Model and Reference Points [6], the Ret-RP is separated into an *Access* part and a *Usage* part. For the Ret-RP, the access part describes how a consumer business role accesses a retailer business role to make use of services provided by service providers; the usage part is outside the scope of this Supplement. As each part is handled independently in the SPFEE specifications, they can be used independently as well. This Supplement further describes the access part only.

7.3.1 Business Roles and Session Roles

As stated in [6], a business role can take different session roles. Two basic session roles are defined: *User* and *Provider*. A specialization is made when dealing with access related interactions. So, the roles become access user and access provider. The session roles and business domains naming conventions are reflected in the naming of the module structure for the IDL specifications (clause 9)

Although the Ret-RP specifications refer to business roles (in conformance to [6]), the applicability of the specifications themselves can be extended to relationships where the same session roles are involved, irrespective of the business roles involved. For example, whenever an access user and an access provider can be defined, the access part of the Ret-RP specifications can be applied. However, the means to extend Ret-RP specifications to contexts other than the consumer/retailer relationship (for example for retailer to retailer federation) are outside of the scope of this Supplement.

7.3.2 Conformance to the Ret Reference Point Specifications

This Supplement provides the necessary guidelines to identify what conformance to Ret-RP means. Conformance does not mean support for all features, but means support for all mandatory features[36] in conformance to the SPFEE specifications, and conformance the SPFEE specifications for optional features if supported.

The Ret-RP is profiled in terms of mandatory and optional interfaces and operations.

Conformance to Ret-RP is claimed separately for Consumer side and Retailer side.

Therefore, for an SPFEE system, the minimum level of compliance to the Ret-RP means support at least for the mandatory interfaces and operations of one of the two sides (consumer or retailer).

---

[36] Here, feature means interface or operation.

In order for two SPFEE systems to interact via the Ret-RP, it is required that one system conforms to the consumer side and the other to the retailer side.

8 Ret-RP Specification

Ret-RP offers the following capabilities:

– initiation of dialogue between the consumer and retailer domains;

– identification of the domains to each other (either domain can remain anonymous dependent on the interaction requested);

– establishment of a secure association between the domains, (an access session);

– set up of the default context for the control and management of usage functionality (service sessions);

– discovery of service[37] offerings;

– listings of access sessions, service sessions and subscribed services;

– initiation of usage between the domains, (starting a service session);

– control and management of service sessions (e.g. stop, suspend, resume, join, notify changes, etc.).

It can be noted that Ret-RP addresses two types of access functionality:

– functionality dedicated to the access session between the consumer and the retailer;

– functionality related to accessing services, and for which the retailer must invoke one or more service providers which support the actual services (the retailer is a pure retailer, solely dedicated to access functionality).

  It is one of the retailer's functionality to invoke possibly more than one service providers in order to fulfill a consumer's request. This takes place in a way totally transparent to the consumer. The retailer performs one or more invocations on one or more service providers and returns a merged result. It is as well transparent to the service provider who must remain unaware that the retailer is possibly performing similar invocations on other service providers. It must be noted that the specification of the Ret-RP and the Ret-SP-RP are sufficiently flexible and generic to ensure that the multiplexing role of the retailer can be performed. Consequently further specification on this matter is considered outside the scope of the specification of the Ret and Ret-SP reference points.

The functions dedicated to accessing services are:

– discovery of service[38] offerings;

– listings of access sessions, service sessions and subscribed services;

– initiation of usage between the domains, (starting a service session);

– control and management of service sessions (e.g. stop, suspend, resume, join, notify changes, etc.).

Ret-RP largely addresses the establishment, and use of a secure association between the domains, termed an Access Session.

---

[37] These services can be primary [e.g. Video on Demand (VoD)], ancillary to the primary (e.g. configuration management for VoD) or administrative (e.g. subscriber management for VoD).

[38] These services can be primary [e.g. Video on Demand (VoD)], ancillary to the primary (e.g. configuration management for VoD) or administrative (e.g. subscriber management for VoD).

The Ret-RP access session is defined in the following sections. Many of the interfaces and operations defined for Ret-RP will be applicable to the access parts of other inter-domain reference points (such as Ret-SP, and Retailer_to_Retailer). In order to facilitate this re-use, a set of interfaces have been defined which can be re-used in other reference points. These interfaces can be recognized by the prefix `i_User` or `i_Provider`. These interface types correspond to the Access User and Access Provider roles.

Interfaces for the Ret-RP are designated with the prefixes: `i_Consumer` and `i_Retailer`. These correspond to the consumer and retailer business administrative domains from the SPFEE Business Model [6]. For Ret-RP, these domains take the access user and access provider roles. All of the `i_Consumer` and `i_Retailer` interfaces are inherited from corresponding `i_User` and `i_Provider` interfaces. Any specialization for the Ret-RP are defined in the `i_Consumer`/`i_Retailer` interfaces. However, no specializations are defined at present.

In summary, the Ret-RP interfaces are inherited from generic user-provider interfaces that can be reused in many other reference points.

NOTE – The main body of this Supplement describes only interfaces and operations on interfaces. A complete listing of the IDLs, and how the interfaces are grouped into modules can be found in clause 9.

The remainder of the Access Session of Ret-RP is structured as follows:

Subclause 8.1, "Overview of Access interfaces for Ret-RP" contains a description of the access interfaces of Ret-RP, together with a short explanation of every operation. It identifies only those interfaces which are exported over Ret-RP.

Subclause 8.2, "User-Provider Interfaces" identifies the generic user-provider interfaces that are not exported over Ret-RP. It shows the inheritance hierarchy for the interfaces exported over Ret-RP. It also describes the generic interfaces so that they can be re-used for other inter-domain reference points.

Subclause 8.3, "Access Information View" gives an information view of Ret-RP. It describes the types of information passed over the Ret-RP.

Subclause 8.5, "Access Interface Definitions: Consumer Domain Interfaces" and 8.6, "Access Interface Definitions: Retailer Domain Interfaces" describe the operations of Ret-RP interfaces supported by the consumer and retailer domains in detail, including parameters and dynamics.

IDL definitions of each of the interfaces can be found in clause 9.

8.1 Overview of Access interfaces for Ret-RP

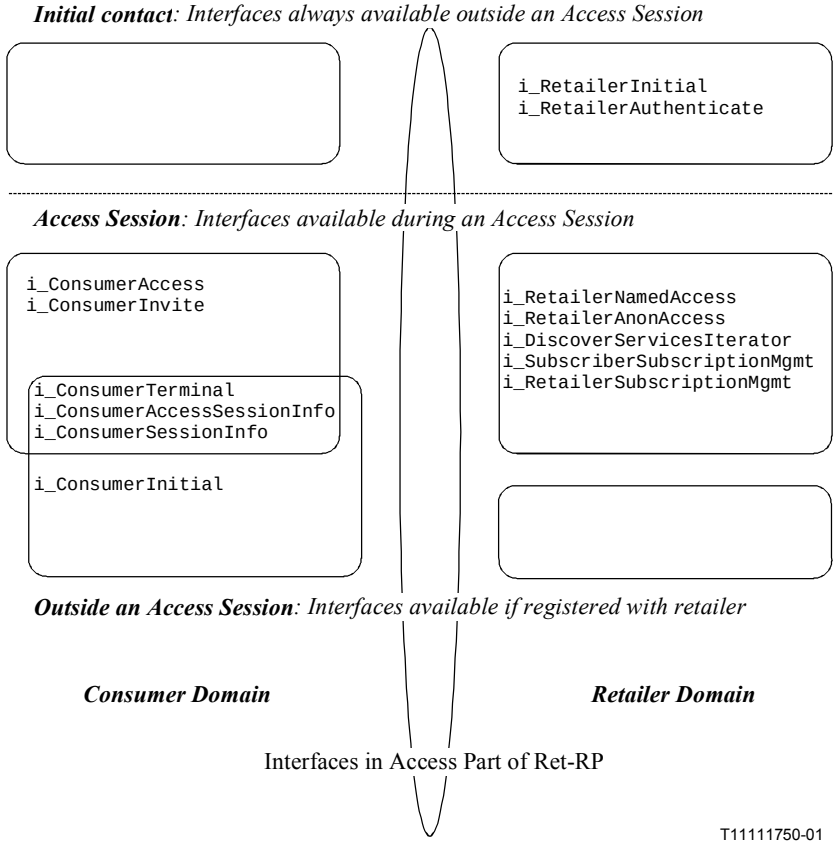
The Access Part of the Ret-RP is defined by a set of interfaces which are offered over the reference point. All of the interfaces in the Access Part are categorized by which side of the RP offers the interface: the consumer, or the retailer.

The interfaces are also categorized by whether they are accessible during an access session; always available outside of an access session; or may be registered to be available outside of an access session.

Registration of interfaces can only be done by the consumer on the retailer domain during an access session. The lifetime of registration depends on how the consumer registers his interfaces, i.e. only as long as the access session exists or permanent.

The following diagram names all of the interfaces defined by the Access Part, and categorizes them as above.

The interfaces: "always available outside of an access session" are supported by the retailer to allow a consumer to request the establishment of an access session. They are the initial point of contact for the consumer, and allow him to authenticate himself and the retailer; establish the access session; and gain a reference to an `i_RetailerNamedAccess`, or `i_RetailerAnonAccess` interface.



The interfaces: "available during an access session" allow the consumer and retailer to interact during an access session. The interfaces on the retailer allow the consumer to discover services; initiate usage of those services; control and manage those services, (e.g. stop, suspend, resume, etc.) and register the consumer's context and interfaces with the retailer. The interfaces supported by the consumer allow the retailer to discover interfaces and terminal configuration of the consumer; notify changes in access and service sessions; and send invitations to join service sessions. These capabilities are only possible during the access session.

The interfaces: "available if registered with retailer" are supported by the consumer. They must be registered with the retailer for use outside the access session to be accessible. With the appropriate interface registered, the retailer is able to perform all of the operations "available during an access session", as well as request the consumer to initiate an access session with them.

The following sections will globally describe the interfaces and their operations. Detailed information about operations, their parameter lists and dynamics can be found in 8.5 and 8.6.

The interfaces dedicated to subscription (i_SubscriberSubscriptionMgmt and i_RetailerSubscriptionMgmt) are described in detail in 8.7, "Subscription Management".

8.1.1 Example Scenario of Access part of Ret-RP

This subclause is an example of the use of the access Ret-RP interfaces. It describes a consumer making use of retailer interfaces to establish an access session; make use of retailer facilities, and register to receive invitations outside of an access session.

1) A consumer domain contacts the retailer by gaining a reference to an i_RetailerInitial interface[39].

2) The consumer domain calls the `requestNamedAccess`() operation on `i_RetailerInitial`, as he wishes to establish an access session with the retailer as a named user. (If the consumer wished to remain anonymous, he could use the `requestAnonymousAccess`() operation on that interface instead.)

   2a) If CORBA security services have been used, then both domain's credentials and other authentication information will have been exchanged, and both consumer and retailer will have been authenticated to each other. The call to `requestNamedAccess`() returns a reference to an `i_RetailerNamedAccess` interface. (An access session has been established between the consumer and retailer domains.)

   2b) If CORBA security services are not used, then the call to `requestNamedAccess`() fails, and an e_AuthenticationError exception is raised. This exception contains a reference to an `i_RetailerAuthenticate` interface, which the consumer can use to authenticate himself. After this, the consumer calls `requestNamedAccess`() on `i_RetailerInitial` in order to gain a reference to the `i_RetailerNamedAccess` interface.

3) At this point, an access session has been established, and the consumer domain has a reference to the `i_RetailerNamedAccess` interface.

4) The consumer domain informs the retailer domain of its interfaces and terminal configuration by calling the **setUserCtxt**() operation on `i_RetailerNamedAccess`. The retailer gains references to the `i_ConsumerAccess`, `i_ConsumerInvite`, `i_ConsumerTerminal`, and `i_ConsumerSessionInfo` interfaces for use within this access session.

5) The consumer can now, by invoking the appropriate operations on the `i_RetailerNamedAccess` interface:

   – discover services offered by the retailer (`discoverServices`());
   – subscribe to those services (by starting a subscription service);
   – list the access sessions and service sessions they are currently involved with (`listAccessSessions`(), `listServiceSessions`());
   – start a new service session (`startService`());
   – suspend, resume, join and end existing sessions (`suspendSession`(), `resumeSession`(), `endSession`());
   – gain references to retailer-specific interfaces (`getInterfaces`());
   – register interfaces for use outside of an access session (`registerInterfaceOutsideAccessSession`());
   – and more...

---

[39] The mechanism by which the consumer gains this interface is not prescribed by Ret-RP.

6) The retailer can:
   - gain references to retailer-specific interfaces (using `i_ConsumerAccess` interface);
   - invite the consumer to join a session (using `i_ConsumerInvite` interface);
   - discover the terminal configuration (using `i_ConsumerTerminal` interface);
   - inform the consumer of changes in their access and service sessions (using `i_UserAccessSessionInfo`, and `i_UserSessionInfo` interfaces).
7) The consumer registers the `i_ConsumerInitial` interface for use outside of an access session, (using `registerInterfaceOutsideAccessSession`() on `i_RetailerNamedAccess`). Then he ends the access session (), and can no longer make requests to the retailer.
8) The retailer can still invite the consumer to join a service session, using `inviteUserOutsideAccessSession`() on the `i_ConsumerInitial` interface.
9) If the consumer wished to join the session they've been invited to, then they would have to establish another access session, (as in step 1.)

8.1.2 Always available outside an Access Session

Only retailer interfaces are always available outside an access session.

The following interfaces are provided by the retailer to allow the consumer and/or retailer to authenticate themselves, and establish an access session.

- `i_RetailerInitial` – This interface is the consumer's initial point of contact for the retailer. It can be used to request the establishment of an access session. The access session provides a consumer access to use his subscribed services, etc. through an i_RetailerNamedAccess, or i_RetailerAnonAccess interface, if the consumer is authenticated as a named, or anonymous user respectively. If the consumer is not authenticated, it returns a reference to the i_RetailerAuthenticate interface, to allow this authentication to occur.
- **`i_RetailerAuthenticate`** – This interface is used by the consumer to authenticate themselves and the retailer and for passing credentials that can be used to establish the access session.

8.1.2.1 i_RetailerInitial Interface

The i_RetailerInitial interface allows the consumer to request the establishment of an access session.

- `requestNamedAccess`() allows the consumer to identify himself to the retailer, and establish an access session. A secure context may have already been set-up between the consumer and the retailer using CORBA security services. In this case, this operation returns a reference to an i_RetailerNamedAccess interface. If the consumer has not already been authenticated, then an e_AuthenticationError exception will be raised. This contains a reference to an i_RetailerAuthenticate interface, which may be used to authenticate and set-up the secure context. Then this operation can be invoked again to retrieve the reference to the i_RetailerNamedAccess interface.
- **`requestAnonymousAccess`**() allows the consumer to establish an access session with the retailer without revealing his identity. The access session will provide access to some services, although the consumer may need to negotiate with the retailer over which services are available. (The services will obviously not be specialized to the consumer.) The consumer interacts with the retailer through an i_RetailerAnonAccess interface. This operation is otherwise the same as requestNamedAccess().

8.1.2.2 i_RetailerAuthenticate interface

The i_RetailerAuthenticate interface allows the consumer and/or the retailer to be authenticated and acquire credentials, to set-up a secure context. The interface provides a generic mechanism for authentication which can be used to support a number of different authentication protocols.

The primary purpose of this interface is to verify for the consumer and retailer that they are indeed talking to the domain they have been told they are talking to. It is not intended to necessarily identify the consumer. (requestNamedAccess() is used to identify the consumer, and provide it access to its services.)

- `getAuthenticationMethods`() provides a list of the authentication methods supported by the retailer.
- **`authenticate`**() allows the consumer to select an authentication method, pass authentication data and request specific credentials that may be used for maintaining a secure context. The retailer then returns its authentication data (if required), challenge data for the consumer to respond using continueAuthentication() (if required), and the requested credentials (if possible). If further authentication protocol is required before credentials are returned then these can be returned by continueAuthentication()
- **`continueAuthentication`**() may be called one or more times after authenticate(). It allows the consumer to respond to the challenge data returned from authenticate() or previous continueAuthentication() call. At the first or subsequent calls of continueAuthentication() credentials requested by the consumer may be returned according to the protocol requirements.

8.1.2.3 Available during an Access Session

Both consumer and retailer interfaces are available during an access session.

The consumer supports the following interfaces for the retailer to use during the access session:

- `i_ConsumerAccess` – The retailer can find out about the interfaces in the consumer domain using this interface. It provides the retailer with interface references to other interfaces in the consumer domain.
- `i_ConsumerInvite` – This interface is used by the retailer to notify the consumer of invitations to join service sessions. The consumer can register an i_ConsumerInitial interface to receive invitations outside an access session.
- `i_ConsumerTerminal` – This interface is used by the retailer within an access session to access terminal configuration information, e.g. applications installed, hardware configuration, (NAPs), etc.
- `i_ConsumerAccessSessionInfo` – This interface is used by the retailer to inform the consumer of state changes to other access sessions which this consumer has with this retailer.
- `i_ConsumerSessionInfo` – This interface is used by the retailer to inform the consumer of state changes to service sessions which this consumer has with this retailer. Information is sent on all service sessions, used through all access sessions with this retailer.

All of the consumer supported may be registered with the retailer for use inside or outside of an access session, through operations on the i_RetailerNamedAccess interface. Other retailer-specific interfaces not defined by Ret-RP can also be registered. Registration of the first three mentioned here is mandatory, using the setUserCtxt() operation on the i_RetailerNamedAccess interface. The lifetime for this particular registration is the same as the lifetime of the access session.

The retailer supports two interfaces for use during access sessions. The consumer will only be given a reference to one of these interfaces. If they have authenticated as a named user and invoked the requestNamedAccess() operation, they will be given a reference to i_RetailerNamedAccess;

otherwise if they have authenticated as an anonymous user, and invoked requestAnonymousAccess(), then they will be given a reference to i_RetailerAnonAccess:

- `i_RetailerNamedAccess` – This interface allows a known consumer to access his subscribed services, start and manage service sessions, etc.
- `i_RetailerAnonAccess` – This interface is used by the retailer to notify the consumer of invitations to join service sessions. The consumer can register an i_ConsumerInitial interface to receive invitations outside an access session.

During an access session, a consumer will have access to one of these interfaces, depending on whether they have authenticated as a named or anonymous user. The current definition of Ret-RP does not allow the change from anonymous to named user in the same access session.

The retailer also supports the following interface:

- `i_DiscoverServicesIterator` – A reference to this interface is returned to the consumer after invoking the discoverServices() operation on either of the interfaces above. It is used to retrieve the remaining service descriptions, which were not returned directly from discoverServices().

8.1.2.4 i_ConsumerAccess interface

The i_ConsumerAccess interface allows the retailer access to the consumer domain, during an access session. It allows the retailer to request references to interfaces supported by the consumer domain. These interfaces include those defined by Ret-RP, as well as other retailer specific interfaces.

- `cancelAccessSession`() – allows the retailer to cancel this access session. After this operation has been invoked, neither consumer nor retailer will make use of the other interfaces. (Interfaces registered for use outside the access session, or interfaces within another access session can still be used.)

This interface inherits the following operations from the i_UserAccess interface, for the retailer to gain references to other interfaces supported by the consumer:

- `getInterfaceTypes`() – allows the retailer to discover all of the interface types supported by the consumer domain.
- **getInterface**() – allows the retailer to retrieve an interface reference, giving the interface type, and properties.
- **getInterfaces**() – allows the retailer to retrieve a list of all the interfaces, supported by the consumer.

This interface is registered with the retailer using the setUserCtxt() operation, and is available for use during the current access session.

8.1.2.5 i_ConsumerInvite interface

The i_ConsumerInvite interface allows the retailer to send invitations to join service session, during an access session. It is only available during an access session to receive invitations. If the consumer wishes to receive invitations outside of an access session, then they must register the i_ConsumerInitial interface for use outside an access session.

- `inviteUser`() – allows the retailer to invite the consumer to join a service session. A session description and sufficient information to join the session is available in the parameter list. The session can only be joined using the joinSessionWithInvitation() operation on the i_RetailerNamedAccess interface.
- `cancelInviteUser`() – allows the retailer inform the consumer that an invitation previously sent to the consumer has been cancelled.

This interface is registered with the retailer using the setUserCtxt() operation, and is available for use during the current access session.

8.1.2.6 i_ConsumerTerminal interface

The i_ConsumerTerminal interface allows the retailer to gain information about the consumer domain's terminal configuration, and applications.

– `getTerminalInfo`() – allows the retailer to retrieve information about the consumer domain's terminal. Information on the terminal id, type, network access points, and user applications can be accessed.

This interface is registered with the retailer using the setUserCtxt() operation, and is available for use during the current access session.

8.1.2.7 i_ConsumerAccessSessionInfo interface

The i_ConsumerAccessSessionInfo interface allows the retailer to inform the consumer of changes of state in other access sessions with the consumer, (e.g. access sessions with the same consumer which are created or deleted). The consumer is only informed about access sessions which they are involved in.

- `newAccessSessionInfo`() – This (oneway) operation is used by the retailer to inform the consumer about a new access session in which the consumer is involved.
- `endAccessSessionInfo`() – This (oneway) operation is used by the retailer to inform the consumer that another access session has ended.
- `cancelAccessSessionInfo`() – This (oneway) operation is used to inform the consumer an access session has been cancelled by the retailer.
- `newSubscribedServicesInfo`() – This (oneway) operation is used by the retailer to inform the consumer that they have been subscribed to some new services.

This interface is not registered with the retailer using the setUserCtxt() operation. Instead the consumer domain must register this interface using the i_RetailerNamedAccess interface. It can be registered for use both inside and outside of an access session.

8.1.2.8 i_ConsumerSessionInfo Interface

The i_ConsumerSessionInfo interface allows the retailer to inform the consumer of changes of state in service sessions which the consumer is involved in. Information operations are invoked whenever a change to the service session affects the consumer, (i.e. the session is suspended), but not when the change does not affect the consumer, (i.e. another party in the session leaves). This interface is informed of changes in all service sessions involving the consumer, and not just those associated with this access session.

The following operations inform the consumer that:

- `newSessionInfo`() – a new service session has been started;
- `endSessionInfo`() – an existing service session has ended;
- `endMyParticipationInfo`() – the consumer's participation in the session has ended;
- `suspendSessionInfo`() – an existing service session has been suspended;
- `suspendMyParticipationInfo`() – the consumer's participation in the service session has suspended;
- `resumeSessionInfo`() – a suspended session has been resumed
- `resumeMyParticipationInfo`() – the consumer's participation in the session has resumed;
- `joinSessionInfo`() – the consumer has joined a service session.

This interface can be registered with the retailer using the setUserCtxt() operation. If so, the interface is available during the current access session only.

It can be registered at any other time with the retailer using the register interface operations on the i_RetailerAccess interface. It can be registered for use both inside and outside of an access session.

8.1.2.9 i_RetailerNamedAccess interface

i_RetailerNamedAccess interface allows a known consumer access to his subscribed services. The consumer uses it for all operations within an access session with the retailer. A reference to this interface is returned when the consumer has been authenticated by the retailer and an access session has been established. It is returned by calling requestNamedAccess() on the i_RetailerInitial interface.

It provides the following operations (which are inherited from i_ProviderNamedAccess interface):

- **setUserCtxt**() – allows the consumer to inform the retailer about interfaces in the consumer domain, and other consumer domain information. (e.g. user applications available in the consumer domain, operating system used, etc.). It should be called immediately after receiving the reference to this interface, or subsequent operations may raise an exception.
- **listAccessSessions**() – allows the consumer in this access session to find out about other access sessions that he has with this retailer: (e.g. A consumer is at work, but has an access session set up at home which runs an active security service session).
- **endAccessSession**() – allows the consumer to end a specified access session, either the current one, or another, found using listAccessSessions(). The consumer can also specify some actions to do if there are active service sessions.
- **getUserInfo**() – gets the consumer's username, and other properties.
- **listSubscribedServices**() – lists the services to which the consumer is subscribed. Scoping of subscribed services can be done using property lists. The operation returns sufficient information for the consumer to start a particular (subscribed)service.
- **discoverServices**() – lists all the services available from the retailer. The consumer can scope the list by supplying some properties that the service should have, and a maximum number to return. A reference to an i_DiscoverServicesIterator interface can be used to retrieve the remaining services.
- **getServiceInfo**() – returns the service information for a particular service (identified in the invocation by its serviceId). Similar information (t_ServiceProperties) can be obtained with the listSubscribedServices or discoverServices, but the getServiceInfo is a simplified version, targeting on a single service, and independently from the subscription status.
- **listRequiredServiceComponents**() – retrieves information on how to download the application software in case of Java applets. The terminalInfo is included as an IN parameter to avoid an explicit call of the getTerminalInfo operation. For example in case of Java applet download, the property list will contain an entry with a name-value pair describing the URL of the Java applet; the name will be "URL" and the value the string value of the URL.
- **listServiceSessions**() – lists the service sessions of the consumer. The request can be scoped by the access session, and session properties, (e.g. active, suspended, service type, etc.).
- **getSession**(Models/InterfaceTypes/Interface/Interfaces)() – all retrieve information on a particular session.
- **listSessionInvitations**() – lists the invitations to join a service session that have been sent to the consumer.
- **listSessionAnnouncements**() – lists the service sessions with have been announced. It can be scoped by some announcement properties.
- **startService**() – allows the consumer to start a service session.

– **endSession**() – allows the consumer to end a service session.
– **endMyParticipation**() – allows the consumer to end his participation in a service session.
– **suspendSession**() – allows the consumer to suspend a service session.
– **suspendMyParticipation**() – allows the consumer to suspend his participation in a service session.
– **resumeSession**() – allows the consumer to resume a service session.
– **resumeMyParticipation**() – allows the consumer to resume his participation in a service session.
– **joinSessionWithInvitation**() – allows the consumer to join a service session, to which he has been invited.
– **joinSessionWithAnnouncement**() – allows the consumer to join a service session, which has been announced.
– **replyToInvitation**() – allows the consumer to reply to an invitation. It can be used to inform the service session to which they have been invited, that they will/will not be joining the session, or to send the invitation somewhere else. (It does not allow the consumer to join the session.)

It also supports the following operations inherited from i_ProviderAccessInterfaces interface. These are useful for accessing retailer specific interfaces:

– **getInterfaceTypes**() – allows the consumer to discover all of the interface types supported by the retailer domain.
– **getInterface**() – allows the consumer to retrieve an interface reference, giving the interface type, and properties.
– **getInterfaces**() – allows the consumer to retrieve a list of all the interfaces, supported by the retailer.
– **registerInterface**() – allows a consumer interface to be registered for use within the current access session. The registrations ends when the access session ends, or when the unregisterInterface() operation is called. An interface index is returned to allow the interface to be unregistered.
– **registerInterfaceOutsideAccessSession**() – allows a consumer to register an interface for use outside an access session. (The interface registered should still be available when no access session exists between the consumer and retailer.)
– **listRegisteredInterfaces**() – allows the consumer to list the interfaces which have been registered by them with the retailer. The list defines which interfaces are registered for use inside an access session, and which for use outside.
– **unregisterInterface**() – allows the consumer to unregister an interface, so that the retailer will not attempt to use that interface, (either inside or outside the access session).

8.1.2.10 i_RetailerAnonAccess interface

The i_RetailerAnonAccess interface allows an anonymous consumer access to the retailer's services. The anonymous consumer uses it for all operations within an access session with the retailer. This interface is returned when the consumer calls requestAnonymousAccess() on the i_RetailerInitial interface.

Currently the operations for this interface are not defined. It will support operations similar to those of the i_RetailerNamedAccess interface.

8.1.2.11 i_DiscoverServicesIterator

The i_DiscoverServicesIterator interface is returned by calls to the discoverServices() operation. This operation is used to retrieve a list of services supported by the retailer which match a set of properties. The list generated by this operation may be too large to return as an out parameter. This interface allows the list to be retrieved in digestible chunks by the consumer. Each call to discoverServices() returns a new instance of this interface.

– **maxLeft**() – The consumer can find out how many unseen services are left.

– **nextN**() – The consumer can indicate that he wants to get information about the next n services.

– **destroy**() – The consumer informs the retailer that the interface is no longer needed.

8.1.3 Available outside an Access Session if Registered

The consumer can register some his interface for use by the retailer outside of the current access session. An interface can be registered using the registerInterfaceOutsideAccessSession() operation on the i_RetailerNamedAccess interface. If registered, the retailer will retain a reference to the interface when the consumer/retailer end the current access session. The retailer can invoke operations on this interface without an access session being present.

The retailer will not use the interface registered until the access session in which it was registered has ended. They will continue to use the interface until the interface is unregistered. If another access session is established, the retailer will still invoke operations on the registered interface, in addition to new interfaces provided as part of the new access session.

– i_ConsumerInitial. This interface allows the retailer to initiate an access session with the consumer. It also allows the retailer to send invitations to the consumer outside of an access session.

– i_ConsumerTerminal. The retailer will use this interface to access terminal configuration information, if necessary. See previous description.

– i_ConsumerAccessSessionInfo. The retailer will use this interface to inform the consumer of changes to any of their access sessions. See previous description.

– i_ConsumerSessionInfo. The retailer will use this interface to inform the consumer of changes in state to any of their service sessions. See previous description.

8.1.3.1 i_ConsumerInitial Interface

The i_ConsumerInitial interface allows the retailer to contact the consumer outside of an access session. It can be used to request that the consumer establish an access session with the retailer; and to invite a user to join a service session.

This interface is only available to the retailer if the consumer has registered it during an access session, (using the registerInterfaceUntilUnregistered() operation, on the i_RetailerNamedAccess interface). It is NOT available through a broker, as the i_RetailerInitial interface is.

The following operations are available:

– `requestAccess`() – allows the retailer to request the consumer to set up an access session.

– `inviteUserWithoutAccessSession`() – allows the retailer to send an invitation to the consumer while he is not involved in an access session with the retailer.

– `cancelInviteUserWithoutAccessSession`() – allows the retailer to cancel an invitation sent to the consumer.

8.2 User-Provider Interfaces

The interfaces defined above are for use over the Ret-RP. The interface names used include the names Consumer and Retailer in order to identify that they are for use over the Ret-RP. The descriptions above include all of the operations that are used over Ret-RP.

Other reference points will want to use similar interfaces as those defined above, for access related activities, (e.g. establishing an access session, starting services, etc.). In order to allow other reference points to re-use interfaces and operations, a set of generic access interfaces are defined. These interfaces support the access session roles as defined in 5.4 of this Supplement. The roles supported are access user and access provider. These interfaces can be recognized by the prefix i_User or i_Provider.

The interfaces defined for use over Ret-RP have been defined above, including all of the inherited operations. All of the i_Consumer and i_Retailer interfaces are inherited from corresponding i_User and i_Provider interfaces. Any specializations for the Ret-RP are defined in the i_Consumer / i_Retailer interfaces. However, no specializations are defined at present.

The figures below define the inheritance hierarchy for both Ret-RP interfaces, and the generic User-Provider interfaces.

8.2.1 User Interfaces

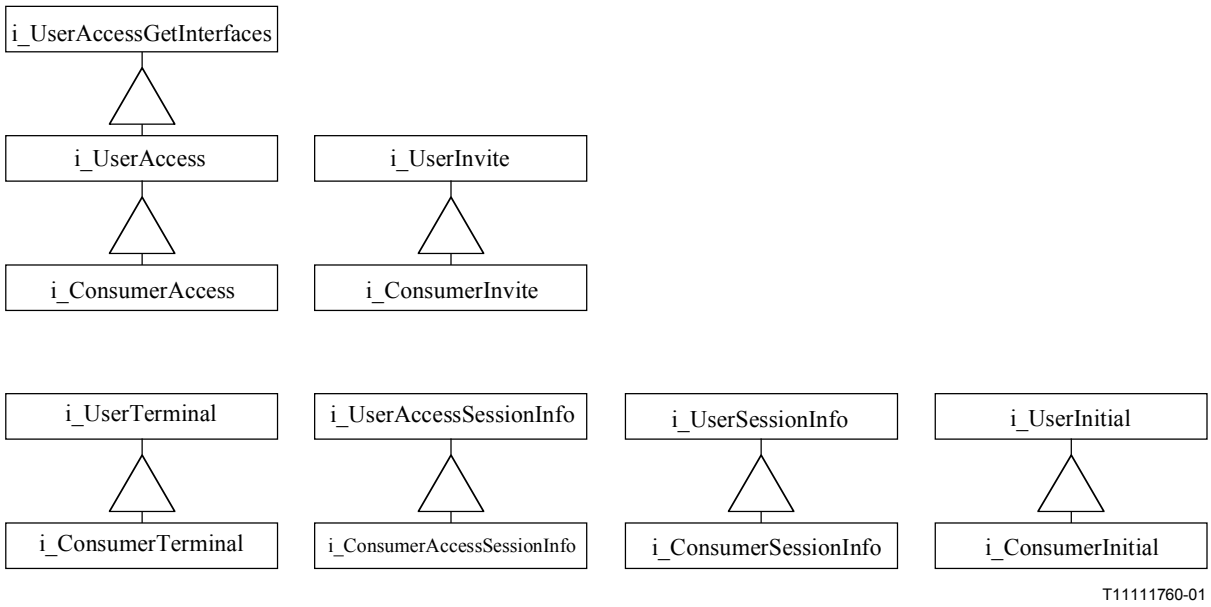


Figure 8-1 – Consumer interfaces inherited from User interfaces

Figure 8-1 shows the consumer interfaces, and the user interfaces that they inherit from. The user and consumer interfaces have a simple mapping, (all the consumer interfaces inherit from a correspondingly named user interface). All of the user interfaces define the operations that are described for the consumer interfaces in 8.1. The only exception to this is the i_UserAccess interface that inherits all of this operation from i_UserAccessGetInterfaces. This is to allow other interfaces to re-use the operations to retrieve interfaces.

8.2.1.1 i_UserAccess

This interface inherits from the abstract interface i_UserAccessGetInterfaces, and defines the following operation:

– **cancelAccessSession**()

8.2.1.2 i_UserInvite

This interface defines the following operations:

– inviteUser()

– cancelInviteUser()

8.2.1.3 i_UserTerminal

This interface defines the following operation:

– getTerminalInfo()

8.2.1.4 i_UserAccessSessionInfo

This interface defines the following operations:

– newAccessSessionInfo()

– endAccessSessionInfo()

– cancelAccessSessionInfo()

– newSubscribedServicesInfo()

8.2.1.5 i_UserSessionInfo

This interface defines the following operations:

– newSessionInfo()

– endSessionInfo()

– endMyParticipationInfo()

– suspendSessionInfo()

– suspendMyParticipationInfo()

– resumeSessionInfo()

– resumeMyParticipationInfo()

– joinSessionInfo()

8.2.1.6 i_UserInitial

This interface defines the following operations:

– requestAccess()

– inviteUserOutsideAccessSession()

– cancelInviteUserOutsideAccessSession()

8.2.2 Provider interfaces

Figure 8-2 shows the retailer interfaces, and the provider interfaces that they inherit from.

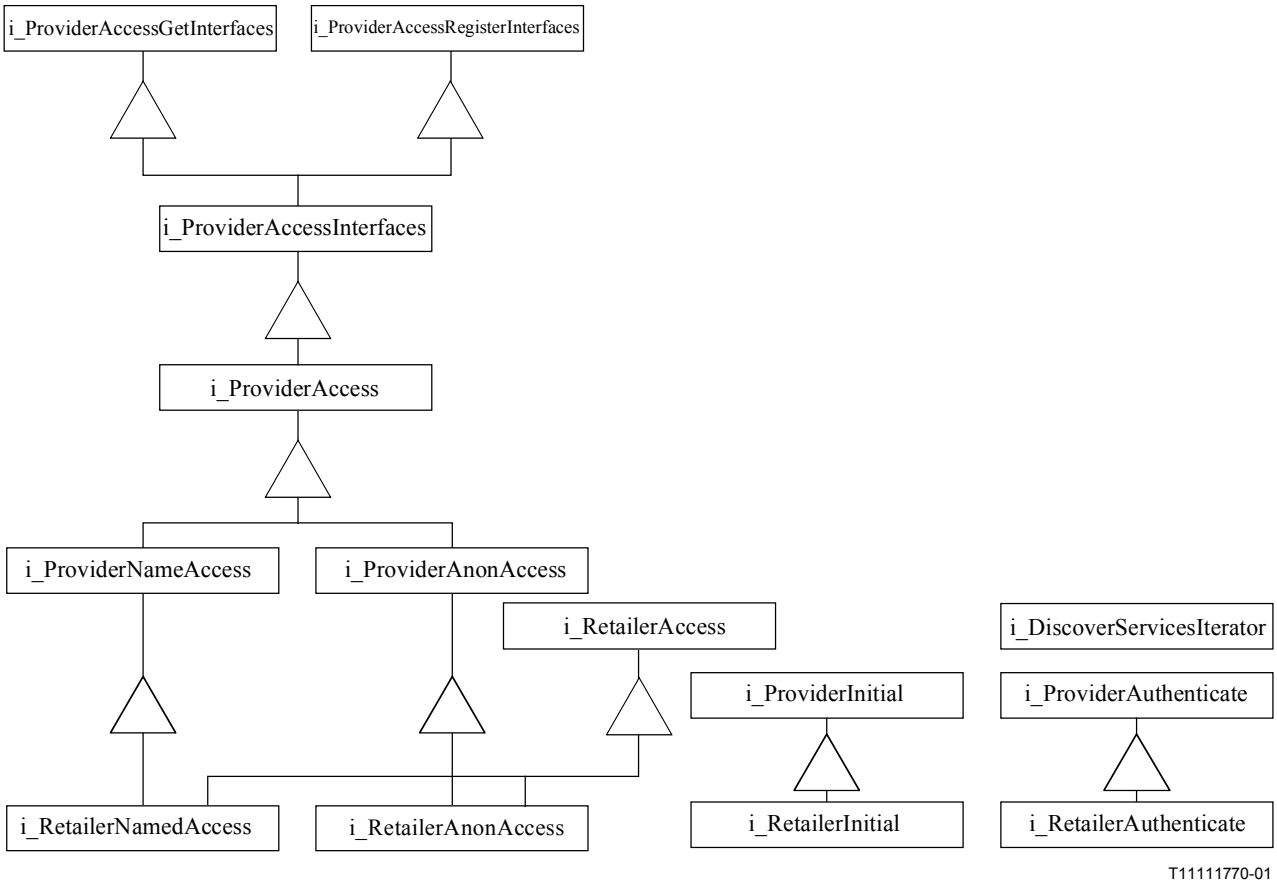


Figure 8-2 – Retailer interfaces inherited from Provider interfaces

The inheritance hierarchy for the i_RetailerNamedAccess and i_RetailerAnonAccess interface is complex. Both inherit from i_RetailerAccess. This interface defines Ret-RP specific operations that are common to both interfaces. (Currently no operations are defined here.)

i_RetailerNamedAccess also inherits from i_ProviderNamedAccess, which defines the operations available for the generic access provider role, where the user domain supports a known user. i_ProviderNamedAccess defines all of the operations offered by i_RetailerNamedAccess.

i_RetailerAnonAccess inherits from i_ProviderAnonAccess, which defines the operations available for the generic access provider role, where the user domain supports an anonymous user. Currently, i_ProviderAnonAccess only inherits operations from i_ProviderAccess.

i_ProviderAccess interface defines the generic access provider role, for reuse in other reference points. It is inherited by both i_ProviderNamedAccess and i_ProviderAnonAccess. Currently, no operations are defined for this interface. In the future, some of the operations defined for i_ProviderNamedAccess will be moved here, as they are common to both interfaces, being appropriate to known and anonymous users.

8.2.2.1 i_ProviderNamedAccess

This interface defines the following operations, and inherits others from i_ProviderAccess:

– setUserCtxt()
– listAccessSessions()
– endAccessSession()
– getUserInfo()
– listSubscribedServices()
– discoverServices()
– getServiceInfo()
– listRequiredServiceComponents()
– listServiceSessions()
– getSession(Models/InterfaceTypes/Interface/Interfaces)()
  listSessionInvitations()
– listSessionAnnouncements()
– startService()
– endSession()
– endMyParticipation()
– suspendSession()
– suspendMyParticipation()
– resumeSession()
– resumeMyParticipation()
– joinSessionWithInvitation()
– joinSessionWithAnnouncement()
– replyToInvitation()

8.2.2.2 i_ProviderAnonAccess

This interface defines no operations. It inherits from i_ProviderAccess.

8.2.2.3 i_ProviderAccess

This interface defines no operations. It inherits from i_ProviderAccessInterfaces.

8.2.3 Abstract interfaces

This subclause describes the abstract interfaces which are inherited in several retailer and consumer interfaces. They are not exported over Ret. The main purpose of these interfaces is to provide a generic mechanism for registration and retrieval of interfaces in a certain domain.

- `i_UserAccessGetInterfaces` – allows the provider to retrieve all interfaces, only interfaces that have certain properties or interface types of the current access session.
- `i_ProviderAccessGetInterfaces` – allows the user to retrieve all interfaces, only interfaces that have certain properties or interface types of the current access session.
- `i_ProviderAccessRegisterInterfaces` – allows the user to register interfaces for the lifetime of an access session or permanent. It also offers an operation to unregister interfaces.
- `i_ProviderAccessInterfaces` – inherits the previous two and does not offer additional functionality.

8.2.3.1 i_UserAccessGetInterfaces

This interface defines the following operations:

– `getInterfaceTypes()`
– **getInterface**()
– **getInterfaces**()

8.2.3.2 i_ProviderAccessGetInterfaces

This interface defines the following operations:

– `getInterfaceTypes()`
– **getInterface**()
– **getInterfaces**()

8.2.3.3 i_ProviderAccessRegisterInterfaces

This interface defines the following operations:

– `registerInterface()`
– `registerInterfaces()`
– `registerInterfaceOutsideAccessSession()`
– `registerInterfacesOutsideAccessSession()`
– `listRegisteredInterfaces()`
– `unregisterInterface()`
– `unregisterInterfaces()`

8.2.3.4 i_ProviderAccessInterfaces

This interface inherits from `i_ProviderAccessGetInterfaces` and `i_ProviderAccessRegisterInterfaces` with no additional operations.

8.3 Common Information View

This subclause describes common types of information, which have a high potential or re-use (in other reference points).

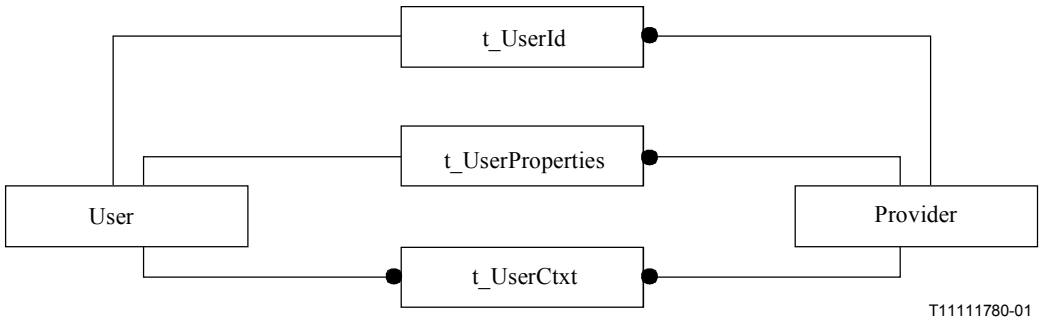


Figure 8-3 – Relationship and Cardinality of common types between User and Provider

8.3.1 Properties and Property Lists

Properties are attributes or qualities of something. In Ret-RP, properties are used to assign a quality to something, or search for those something that have that particular quality.

The something's for Ret-RP can be users, services, sessions, interfaces, etc. Each of these will have different properties, and each property may have a range of different values and structures. (Also for some it is now clear what properties will be defined for them, and some properties will be retailer-specific.)

With this in mind, the type `t_Property` has been chosen to represent a property. Its IDL definition is taken from the CORBA Object Service for Trading, and copied into the SPFEECommonTypes module.

```
// module SPFEECommonTypes
typedef string t_PropertyName;
typedef sequence<t_PropertyName> t_PropertyNameList;
typedef any t_PropertyValue;
struct t_Property {
       t_PropertyName name;
       t_PropertyValue value;
};
typedef sequence<t_Property> t_PropertyList;

enum t_HowManyProps {none, some, all};
union t_SpecifiedProps switch (t_HowManyProps) {
       case some: t_PropertyNameList prop_names;
       case none:
       case all: octet dummy;
};

typedef string Istring;
```

As can be seen above, the `t_Property` is a structure consisting of a name, and a value. The name is an international string, and the value is an `any`. This format allows the recipient of the property to read the string, and match it against the properties they know about. If it is a property they know, then they will also know the format of the value. If they do not know the property, then they should not read the value. (The `any` value contains a typecode that can be looked-up in the interface repository to find the type of the value, but this should be unnecessary most of the time.)

The `t_Property`, and `t_PropertyList` are used to attribute qualities to entities, when we do not wish to define what all of those qualities are at present. (Some of these qualities may also be retailer-specific, and so they can also use these types to extend the Ret-RP.)

For example, some characteristics about the terminal are sent to the retailer after an access session is established. The type `t_TerminalProperties` is defined as a property list to allow these characteristics to be sent to the retailer. It is not clear precisely what characteristics need to be sent to the retailer for all types of terminal, and some may need to send different information than others.

Ret-RP defines a particular property for `t_TerminalProperties`, named: `"TERMINAL INFO"` which has a value of type `t_TerminalInfo`. `t_TerminalInfo` is a structure that holds some information on terminal characteristics. When the retailer reads the `t_TerminalProperties`, and finds a `t_Property` with the name `"TERMINAL INFO"`, then it can look at the value to find the terminal characteristics. The value will still be of type `any`, but is formatted with the information in the `t_TerminalInfo` structure.

However, `t_TerminalInfo` may not be complete, or relevant for all types of terminal. If it does not contain sufficient information, then a future release of Ret-RP, or a retailer, can define another property, e.g. named `"ADDITIONAL TERMINAL INFO"`, with an appropriate value format, to contain

the extra characteristics. The retailer will then receive both properties in the t_TerminalProperties list.

If t_TerminalInfo contains irrelevant information, a retailer can define an entirely different property, and the consumer should send that instead of the "TERMINAL INFO" property.

Ret-RP defines property names and values where it is possible to do so. For some property lists, e.g. t_InterfaceProperties, it is up to the consumer/retailer to determine properties that can be associated with it.

```
// module SPFEECommonTypes
enum t_WhichProperties {
       NoProperties,
       SomeProperties,
       SomePropertiesNamesOnly,
       AllProperties,
       AllPropertiesNamesOnly
};

struct t_MatchProperties {
       t_WhichProperties whichProperties;
       t_PropertyList properties;
};
```

t_MatchProperties is used to scope the return values of some operations. These operations return lists of something. t_MatchProperties is used to identify which something to return, based on the something's properties. (E.g. for the operation listSubscribedServices, the something's are a consumer's subscribed services. The t_MatchProperties parameter defines the properties of the subscribed services which are to be returned in the list.)

t_MatchProperties contains a t_PropertyList, and an enumerated type t_WhichProperties. The t_PropertyList contains the properties which need to be matched. The t_WhichProperties identifies whether some, all or none of the properties must be matched, and whether the property name and value, or just the property name must be matched.

For example, in the operation listSubcribedServices, if t_WhichProperties is:

NoProperties, then the subscribed services don't have to match any properties, and so all subscribed services are returned.

SomeProperties, then the subscribed services must match at least one property in the t_PropertyList, (both the property name and value must match), to be included in the returned list.

SomePropertiesNamesOnly, then the subscribed services must match at least one property name in the t_PropertyList to be returned. The values of the properties in the t_PropertyList may not be meaningful, and should not be used.

AllProperties, then the subscribed services must match all the properties in the t_PropertyList, (both the property name and value must match), to be included in the returned list.

AllPropertiesNamesOnly, then the subscribed services must match all the property names in the t_PropertyList to be returned. The values of the properties in the t_PropertyList may not be meaningful, and should not be used.

8.3.2 User Information

```
// module SPFEECommonTypes
typedef Istring t_UserId;
typedef Istring t_UserName;
typedef t_PropertyList t_UserProperties;
```

t_UserId identifies the user to the retailer. It is unique to this user within the scope of this retailer. It is used in requestNamedAccess(), and is returned by getUserInfo(). The t_UserId does NOT contain the name of the retailer, and so cannot be used to contact the retailer. It may be sent to a broker/naming service when attempting to contact a retailer along with the retailer name.

t_UserProperties is a sequence of t_UserProperty. It contains information about the user, that they wish to pass to the retailer. The following property names are defined for t_UserProperty. Other property names are allowed, but are retailer specific.

```
// Property Names defined for t_UserProperties:
// name:   "PASSWORD"
// value:  string
// use:    user password, as a string.

// name:   "SecurityContext"
// value:  opaque
// use:    to carry a retailer specific security context
//     e.g. could be used for an encoded user password.
```

8.3.2.1 e_UserDetailsError Exception

```
// module SPFEECommonTypes
enum t_UserDetailsErrorCode {
       InvalidUserName,
       InvalidUserProperty
};

exception e_UserDetailsError {
       t_UserDetailsErrorCode errorCode;
       t_UserName name;
       t_PropertyErrorStruct propertyError;
};
```

The e_UserDetailsError exception is defined for operations which require a t_UserDetails parameter. (e.g. inviteUserReq() on usage part of Ret-RP). The exception is raised if the t_UserName, or the t_UserProperties are invalid.

The following error codes can be used to define the problem encountered:

– InvalidUserName:

  The t_UserName parameter does not contain a valid party identifier. (This can be because the t_UserName is wrongly formatted, or the t_UserName given does not refer to any known user.)

  The t_UserName name variable in the exception contains the value of the t_UserName parameter passed in the operation invocation.

– InvalidUserProperty:

  The t_UserProperties parameter is in error. The propertyError element of the exception describes the type of error in the user property. If the propertyError contains InvalidPropertyName, then the property name is not legal for this operation. If it contains InvalidPropertyValue, then the value is not a legal value for the property name. If the propertyError contains UnknownPropertyName, then the session does not recognize the property name. Some sessions may ignore t_PropertyName's that they do not recognize. They should not process t_PropertyValue associated with the t_PropertyName but may process the other t_Property's in the t_UserProperties parameter. Such sessions do not need to raise the exception with this error code.

8.3.3 User Context Information

```
// module SPFEECommonTypes
typedef Istring t_UserCtxtName;

// module SPFEEProviderAccess
struct t_UserCtxt {
    SPFEECommonTypes::t_UserCtxtName                    ctxtName;
    SPFEEAccessCommonTypes::t_AccessSessionId          asId;
    Object accessIR;                                 // type: i_UserAccess
    Object terminalIR;                               // type: i_UserTerminal
    Object inviteIR;                                 // type: i_UserInvite
    Object sessionInfoIR;                            // type: i_UserSessionInfo
    SPFEEAccessCommonTypes::t_TerminalConfig          terminalConfig;
};
```

`t_UserCtxt` informs the retailer about the consumer domain, including the name of the context, interfaces available during this access session, and terminal configuration information The `t_UserCtxt` is only used within the access part of Ret-RP, but is included here to aid the read in understanding the `t_UserCtxtName`. A full description is given in 8.4.

`t_UserCtxtName` is a name given to this consumer context. It is generated by the consumer domain. It is used to distinguish between access sessions to different consumer domains/terminals. When listing the access sessions, the `t_UserCtxtName` is returned, (along with the t_AccessSessionId), as the former should be a more human readable name for the 'terminal' that the access session is connected to.

8.3.4 Usage related types

8.3.4.1 t_SessionId

```
// module SPFEECommonTypes
typedef unsigned long t_SessionId;
```

All operations on the party domain interfaces, (incl. all Exe's and Info's) include a `t_SessionId` parameter.

This allows the party domain to identify the service session sending each operation request. It is long (32 bits). The `t_SessionId` is the same as the sessionId provided by the `startService`(), or `joinSession`() operation for this session. (i.e. the id appearing on the listSessions list in the access part matching this `t_SessionId` will refer to the same session.) If the party domain does not recognize the `t_SessionId`, it may raise a `PD_InvalidSessionId` error code in the `e_PartyDomainError` exception.

8.3.4.2 t_ParticipantSecretId

```
// module SPFEECommonTypes
typedef sequence<octet, 16> t_ParticipantSecretId;
```

All operations on the provider domain interfaces of the service session, (incl. all Requests) include a `t_ParticipantSecretId` parameter. This type is also returned by requests to start, and join a service session.

This allows a service session to identify the sender of each operation request. It is a 128-bit key. The format of the key is not defined, other than all zeros assumes the participant does not know or does not require a key. The session may raise an `InvalidParticipantSecretId` error code in the `e_UsageError` exception, if a key is necessary to make a request.

The `t_ParticipantSecretId` is provided so that sessions can be implemented using only a single interface for all the participants. The session can still be reasonably assured that the request comes from the identified user, and not a different user.

It is not intended that the t_ParticipantSecretId is used as the primary security mechanism. CORBA security, or other security contexts should still be used to underlie the party domain-provider domain interactions.

8.3.5 Invitations and Announcements

Invitations allow a session to ask a specific end-user to join a 'running' session. Invitations are delivered to the consumer domain for the end-user, if an access session exists. If no access session exists with the consumer domain, the invitation may be delivered to a 'pre-registered' interface, or stored until an access session is established. They contain sufficient information for the user to: identify the user that requested the invitation be sent; find and join the session, or refuse. (All of these operations are defined across the access part of Ret-RP, and the retailer is always involved in allowing the consumer to find and join the session.)

```
// module SPFEEAccessCommonTypes
typedef unsigned long t_InvitationId;
typedef SPFEECommonTypes::Istring t_InvitationReason;

struct t_InvitationOrigin {
       SPFEECommonTypes::t_UserId              userId;
       SPFEECommonTypes::t_SessionId           sessionId;
};

struct t_SessionInvitation {
       t_InvitationId                          id;
       SPFEECommonTypes::t_UserId              inviteeId;
       t_SessionPurpose                        purpose;
       t_ServiceInfo                           serviceInfo;
       t_InvitationReason                      reason;
       t_InvitationOrigin                      origin;
       SPFEECommonTypes::t_PropertyList        invProperties;
};

typedef sequence<t_SessionInvitation> t_InvitationList;

// module SPFEECommonTypes
enum t_InvitationReplyCodes {
       SUCCESS, UNSUCCESSFUL, DECLINE, UNKNOWN, ERROR,
       FORBIDDEN, RINGING, TRYING, STORED, REDIRECT, NEGOTIATE,
       BUSY, TIMEOUT
};

typedef t_PropertyList t_InvitationReplyProperties;

struct t_InvitationReply {
       t_InvitationReplyCodes reply;
       t_InvitationReplyProperties properties;
};
```

t_SessionInvitation describes the service session to which the consumer has been invited, and provides a t_InvitationId to identify this invitation when joining. (It does not give interface references to the session, nor any information which would allow the consumer to join the session without first establishing an access session with this retailer.) It also provides a t_UserId with the id of the invited user. The consumer domain can check that the invitation is for an 'end-user' that is known to this domain.

t_SessionPurpose is a string describing the purpose of the session. A session purpose may be defined when the session is started (through t_StartServiceSSProperties), or during the session.

t_ServiceInfo is the subscribed service that the consumer can use to join the session. It is described in 8.4.

t_InvitationReason is a string describing the reason that this invitation was sent to the invited user. It can be defined by the party which requested the invitation, or by the session.

t_InvitationOrigin is a structure defining where the invitation has been generated. It contains the userId of the user that started the session, and their session id for the session.

A t_InvitationReply is returned which allows the consumer to inform the retailer of the action they will take regarding the invitation. The following reply codes are defined:

- SUCCESS – the consumer agrees to join the service session. (The consumer will need to establish an access session before they can join the service session. (This does NOT have to be established from the terminal that received the invitation.) They will then use joinSessionWithInvitation() on the i_RetailerNamedAccess interface to join the session.) The consumer can use replyToInvitation() to 'change their mind', and not join the session, but they should really have replied with RINGING, or another reply code rather than SUCCESS.
- UNSUCCESSFUL – the consumer couldn't be contacted through this operation. (They will not be joining the session due to this invitation. However, if the same invitation was sent to multiple interfaces, a reply from another interface may indicate that the consumer will join the session.)
- DECLINE – the consumer declines to join the session.
- UNKNOWN – the consumer that has been sent the invitation is not known by this interface. (The t_SessionInvitation contains a t_UserId to allow the consumer domain to check the invitation is for a user known to this domain.)
- FAILED – the consumer is unable to join the service session. (No reason is given. The invitation may be badly formatted, or the consumer may be unable to join sessions.)
- FORBIDDEN – the consumer domain is not authorized to accept the request.
- RINGING – the consumer is known by this domain and is being contacted. The retailer should not assume that the consumer will join the session. (If the consumer wishes to join the session then they can do so as describe in SUCCESS above. If they wish to inform the retailer about their status regarding this invitation, they can use replyToInvitation() on the i_RetailerNamedAccess interface.)
- TRYING – the consumer is known by this domain, but cannot be contacted directly. The consumer domain is performing some action to attempt to contact the consumer. The retailer can treat this as RINGING.
- STORED – the consumer is known by this domain, but is not being contacted at present. The invitation has been stored for retrieval by the consumer. (The retailer can treat this as RINGING, although it may be awhile before the consumer responds.).
- REDIRECT – the consumer is known by this domain, but they are not available through this interface. The retailer should use the address given in t_InvitationReplyProperties, to contact the consumer.
- NEGOTIATE – the consumer is known by this domain, but they are not being contacted at present. The t_InvitationReplyProperties contains a set of alternatives that the retailer could try in order to contact the consumer. (These alternatives are not defined by Ret RP, and so are retailer specific at present.)
- BUSY – the consumer cannot be contacted because they are 'busy'. This code should be treated as for UNSUCCESSFUL.

- `TIMEOUT` – the consumer cannot be contacted, as the consumer domain has timed out while trying to contact them. i.e. the consumer domain has a time out value for contacting the consumer using the method for contacting them, (e.g. pop-up window, ringing phone), and this time has expired. This code should be treated as for `UNSUCCESSFUL`.

These invitation reply codes have been taken from the Internet Engineering Task Force working group, Multimedia Multiparty Session Control (MMUSIC) draft standard 'Session Initiation Protocol'.

Announcements allow a session to publicize itself to a 'group' of end-users. The announcements are not directed to a specific user, nor are they 'delivered' to the end-user. Announcements are stored by the retailer domain until the consumer domain requests for a list of announcements. Announcements are return to the consumer, depending upon the 'groups' to which the user belongs. (These are defined by user properties, but no specific mechanism for defining announcement groups has been specified by Ret-RP. Announcements contain sufficient information for the user to join the session. (This operation is defined across the access part of Ret-RP, and the retailer is always involved in allowing the consumer to find and join the session.)

Draft definition: The structure for announcements is draft only.

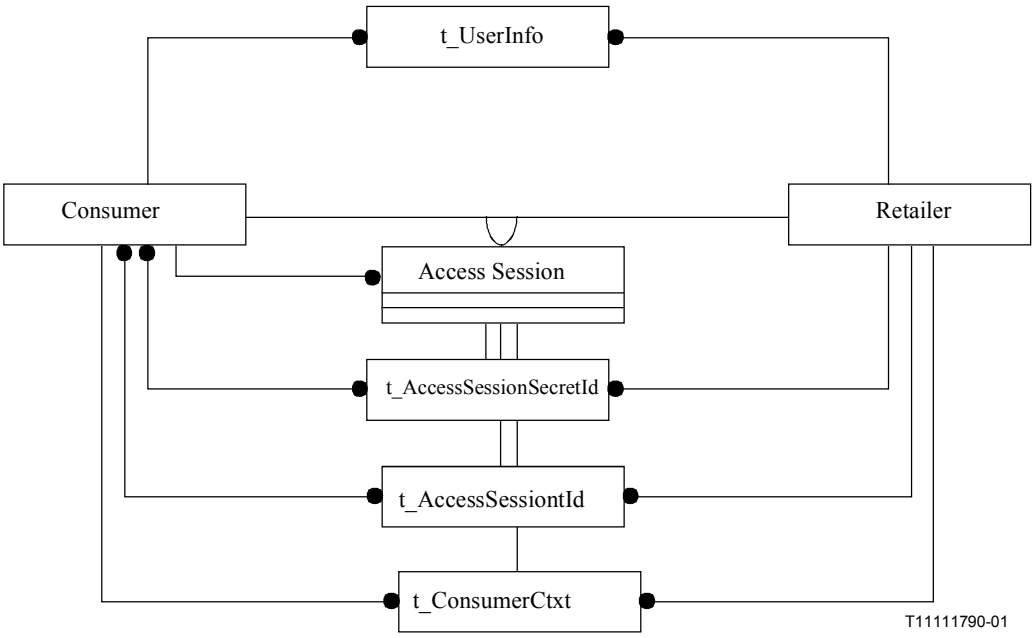
```
// module SPFEECommonTypes
typedef t_PropertyList t_AnnouncementProperties;

struct t_SessionAnnouncement {
       t_AnnouncementId announcementId;
       t_SessionPurpose sessionPurpose;
       t_ServiceInfo serviceInfo;
       t_AnnouncementProperties properties;
};

typedef sequence<t_SessionAnnouncement> t_AnnouncementList;

// module SPFEEAccessCommonTypes
typedef unsigned long t_AnnouncementId;
```

`t_SessionAnnouncement` describes the session that is being announced, and the 'group' of users that the announcement is broadcast to. It is a structure containing the announcementId, the sessionPurpose, the serviceInfo, and a list of announcement properties. No property names or values are defined by Ret-RP for announcements. The announcement properties allow the retailers to define their own types for announcements, which can be passed using the announcement operations defined by Ret-RP.

`t_AnnouncementId` identifies an announcement to the consumer domain. The consumer domain can request a list of announcements which are associated with this end-user. The `t_AnnouncementId` is used by the consumer domain to distinguish between the announcements it receives. The ids for each announcement can only be used by this user. They do not uniquely identify the announcement for all consumers of a retailer.

8.4 Access Information View

This subclause describes the types of information passed across the Ret-RP. The types are passed in operations defined in the access interfaces.



8.4.1 Access Session Information

```
// module SPFEEAccessCommonTypes
typedef unsigned long t_AccessSessionId;
typedef sequence<octet, 16> t_AccessSessionSecretId;
```

t_AccessSessionId is used to identify an access session. The t_AccessSessionId for the consumer's current access session is returned by requestNamedAccess() or requestAnonAccess(). The t_AccessSessionId for other access sessions can be found using listAccessSessions() on the i_RetailerNamedAccess interface. (Anonymous users can only have a single access session, and so only a single t_AccessSessionId). The t_AccessSessionId is scoped by the consumer, i.e. for a single consumer (t_UserId) all t_AccessSessionId's are unique.

t_AccessSessionSecretId is used to identify within which access session a request on the requestNamedAccess() is made. Each access session of a consumer has a unique t_AccessSessionSecretId. It is returned by requestNamedAccess().

All operations on requestNamedAccess() take a t_AccessSessionSecretId as their first parameter. This parameter can be checked by the retailer to determine within which access session of the consumer the request originated. This is useful when the behaviour of the request is dependant on the consumer context, (e.g. startService() checks the consumer context to determine if this service can be used).

t_AccessSessionSecretId is only known within the access session it is created. It is not known to other access sessions of the same consumer, and is not available through listAccessSessions(). This is because it is being used to determine the access session within which the request is made. If another access session gained the t_AccessSessionSecretId of this access session, then it could use it to pretend the request came from this access session. For this reason, it should not be displayed to a human consumer, or other applications in the consumer domain. t_AccessSessionSecretId is not itself a security mechanism, as CORBA security is still needed to set-up security contexts between the consumer and retailer domains. However, it does allow the retailer to easily discover the sender of a particular request.

8.4.2 User Information

Most of the user related information is described in the common types section (see 8.3).

```
// module SPFEEAccessCommonTypes
struct t_UserInfo {
       SPFEECommonTypes::t_UserId userId;
       SPFEECommonTypes::t_UserName name;
       SPFEECommonTypes::t_UserProperties userProperties;
};
```

`t_UserInfo` describes the user. It contains the `t_UserId`, the user's name, and `t_UserProperties`. It is returned by `getUserInfo()` on the `i_ProviderAccess` interface.

8.4.3 User Context Information

```
// module SPFEECommonTypes
typedef Istring t_UserCtxtName;

// module SPFEEProviderAccess
struct t_UserCtxt {
       SPFEECommonTypes::t_UserCtxtName ctxtName;
       SPFEEAccessCommonTypes::t_AccessSessionId asId;
       Object accessIR;                  // type: i_UserAccess[40]
       Object terminalIR;                // type: i_UserTerminal
       Object inviteIR;                  // type: i_UserInvite
       Object sessionInfoIR;             // type: i_UserSessionInfo
       SPFEEAccessCommonTypes::t_TerminalConfig terminalConfig;
};
```

`t_UserCtxt` informs the retailer about the user and the consumer domain, including the name of the context, interfaces available during this access session, and terminal configuration information

`t_UserCtxtName` is a name given to this consumer context. It is generated by the consumer domain. It is used to distinguish between access sessions to different consumer domains/terminals. When listing the access sessions, the `t_UserCtxtName` is returned, (along with the t_AccessSessionId), as the former should be a more human readable name for the 'terminal' that the access session is connected to.

`accessIR` is a reference to the `i_UserAccess` interface supported by the consumer domain for use in this access session.

`terminalIR` is a reference to the `i_UserTerminal` interface supported by the consumer domain for use in this access session.

`inviteIR` is a reference to the `i_UserInvite` interface supported by the consumer domain for use in this access session.

All of the preceding three interface references should be set to valid interfaces in the consumer domain.

`sessionInfoIR` is a reference to the `i_UserSessionInfo` interface supported by the consumer domain for use in this access session. It is not necessary to supply a reference for this interface.

`t_TerminalConfig` is a structure containing the terminal id and type; the network access point id and type; and a list of terminal properties. Two property types have been defined `t_TerminalInfo`, described below, and t_ApplicationInfoList, a list of the user applications on the terminal.

---

[40] The type written in the IDL of this parameter is the base class type. The actual type will depend on the reference point used. In this case the retailer can expect the i_Consumer<> type. See also remark at requestNamedAccess(), output namedAccessIR.

t_TerminalInfo gives details on the type of terminal, operating system, etc.

```
// module SPFEEAccessCommonTypes
struct t_TerminalInfo {
       t_TerminalType terminalType;
       string operatingSystem;              // includes the version
       SPFEECommonTypes::t_PropertyList networkCards;
       SPFEECommonTypes::t_PropertyList devices;
       unsigned short maxConnections;
       unsigned short memorySize;
       unsigned short diskCapacity;
};
```

Draft definition: This structure is draft only and is currently under review (new version will be provided for the revised answer).

t_TerminalType is an enumerated type, giving the type of the terminal.

operatingSystem provides the operating system type and version as a string.

networkCards and devices are property lists of the physical devices of the terminal. The property names and values are not defined at present, so their use is retailer specific.

maxConnections is the maximum number of network connections which can be supported by the terminal.

memorySize is the amount of RAM in Megabytes.

diskCapacity is the amount of disk storage in Megabytes.

8.4.4 Service and Session Information

```
// module SPFEEAccessCommonTypes
struct t_ServiceInfo {
       t_ServiceId id;
       t_UserServiceName name;
       t_ServiceProperties properties;
};
```

t_ServiceInfo is a structure which describes a subscribed service of the consumer.

t_ServiceId is the identifier for the service. t_ServiceId is unique among all the consumer's subscribed services. (Other consumer's may be subscribed to the same service, but will have a different t_ServiceId.) The t_ServiceId value persists for the lifetime of a subscription.

t_UserServiceName is the name of the service as a string. The name is chosen by the subscriber when they subscribe to the service. It is the name of the service displayed to the user.

t_ServiceProperties is a property list, which defines the characteristics of this service. They can be used to search for types of service with the same characteristics, e.g. using discoverServices() on i_RetailerNamedAccess.) Currently, no properties have been defined for t_ServiceProperties, and so its use is retailer specific.

```
// module SPFEEAccessCommonTypes
struct t_SessionInfo {
       SPFEECommonTypes::t_SessionId id;
       t_SessionPurpose purpose;
       SPFEECommonTypes::t_ParticipantSecretId secretId;
       SPFEECommonTypes::t_PartyId myPartyId;
       t_UserSessionState state;
       SPFEECommonTypes::t_InterfaceList itfs;
       SPFEECommonTypes::t_SessionModelList sessionModels;
       SPFEECommonTypes::t_SessionProperties properties;
};
```

`t_SessionInfo` is a structure, which contains information which allows the consumer domain to refer to a particular session, when using interfaces within an access session (e.g. `i_RetailerNamedAccess`). It also contains information for the usage part of the session, including the interface references to interact with the session.

`id` is the identifier for this session. It is unique to this session, among all sessions that this consumer interacts with through this retailer. (i.e. if the consumer interacts with multiple retailers concurrently, then they may return `t_SessionId`'s which are identical.)

`purpose` is a string containing the purpose of the session. This may have been defined when the session is created, or subsequently by service specific interactions.

`secretId` is an identifier that the consumer must use when interacting with interfaces on the session which are defined by the SPFEE Session Model. (See Usage Part of the Ret-RP for more details.)

`myPartyId` is the party identifier of this consumer. If the session is using the SPFEE Session Model, with the MultipartyFS feature set, then this identifier will be used to identify this party. The `t_PartyId`'s of other parties in the session are also available through MultipartyFS interfaces.

`state` is the session state as perceived by this consumer. It can be: `UserUnknownSessionState`, `UserActiveSession`, `UserSuspendedSession`, `UserSuspendedParticipation`, `UserInvited`, `UserNotParticipating`. But as the session has just been started, it is likely to be `UserActiveSession`.

`itfs` is a list of interface types and references supported by the session. (It may include service specific interfaces for the consumer to interact with the session.

`sessionModels` is a list of the session models and feature sets that are supported by the session. It may include interface references to interfaces supported for each feature set.

`properties` is a list of properties of the session. Its use is retailer specific.

8.5 Access Interface Definitions: Consumer Domain Interfaces

This subclause describes in detail all the consumer interfaces supported over Ret RP. Each interface and all its supported operations are defined.

Many of the operations are inherited from other interfaces. However, they are described here as though they were defined on the Ret RP specified interfaces. Only the interfaces defined here must be supported for the Ret RP. It is not necessary to support the same inheritance hierarchy as defined previously in 8.2.

These are the interfaces supported by the consumer domain, which are available across the Ret RP:

– i_ConsumerInitial
– i_ConsumerAccess
– i_ConsumerInvite
– i_ConsumerTerminal
– i_ConsumerSessionInfo
– i_ConsumerAccessSessionInfo

8.5.1 i_ConsumerInitial Interface

```
// module SPFEERetConsumerInitial
interface i_ConsumerInitial  :
SPFEEUserInitial::i_UserInitial
{
};
```

This interface is provided to allow a Retailer initiate an access session with the consumer. It also allows the consumer to receive invitations outside of an access session.

The purpose of this interface is to provide an initial contact point for the retailer wishing to contact the consumer. (So its purpose is similar to that of `i_RetailerInitial` interface.) However, this interface is only available to a retailer if the consumer had previously registered the interface for use outside an access session. This is achieved using `registerInterfaceOutsideAccessSession` operation on the `i_RetailerNamedAccess` interface.

The operations described in the following sections are all inherited into this interface from the `i_UserInitial` interface, which supports the generic user-provider roles. No Ret RP specific specializations are defined for this interface.

The following operation signatures are taken from the module `SPFEEUserInitial`. All unscoped types need to scoped by `SPFEEUserInitial::` when used by clients of the `i_ConsumerInitial` interface.

8.5.1.1 requestAccess()

```
void requestAccess (
       in t_ProviderId providerId,
       out t_AccessReply reply
);
```

Draft definition: This operation is draft only.

This operation allows the retailer to request that an access session be established between the consumer and the retailer.

This operation only allows the retailer to request that an access session be established with the consumer. It does not allow the access session to actually be established. In order to set up an access session the consumer must contact the retailer, using the `i_RetailerInitial` interface, and request that an access session is established.

The retailer passes his `t_ProviderId` to the consumer. The consumer uses this to contact the provider, and gain a reference to an `i_RetailerInitial` interface.

The `t_AccessReply` parameter allows the consumer to inform the retailer of the action they will take in response to the request. The following reply codes are defined:

- `SUCCESS` – the consumer agrees to establish an access session. (The consumer will establish the access session as described above.)
- `DECLINE` – the consumer declines to initiate an access session.
- `FAILED` – the consumer is unable to establish an access session.
- `FORBIDDEN` – the consumer domain is not authorized to accept the request.

8.5.1.2 inviteUserAccessSession()

```
void inviteUserOutsideAccessSession (
       in t_ProviderId providerId,
       in SPFEEAccessCommonTypes::t_SessionInvitation invitation,
       out SPFEECommonTypes::t_InvitationReply reply
);
```

Draft definition: This operation is draft only.

Specifically, the `t_SessionInvitation` and `t_InvitationReply` parameters are defined according to the Internet Engineering Task Force working group, Multimedia Multiparty Session Control (MMUSIC) draft standard 'Session Initiation Protocol'.

This operation allows a retailer to send an invitation to join a service session, to a consumer that is not involved in an access session.

This operation is used if the consumer has previously registered this interface for use outside of an access session, and the consumer is not currently in an access session. If the consumer is in an access session with this retailer, then invitations will not be sent using this operation, but will be delivered to the `i_ConsumerAccess` interface involved in the access session.

In order to join the service session described by the invitation, the consumer must establish an access session with the retailer, and use `joinSessionWithInvitation`() operation on the `i_RetailerNamedAccess` interface. The service session cannot be joined without an access session with the retailer.

`t_ProviderId` identifies the retailer to the consumer.

`t_SessionInvitation` describes the service session to which the consumer has been invited, and provides a `t_InvitationId` to identify this invitation when joining. (It does not give interface references to the session, nor any information which would allow the consumer to join the session without first establishing an access session with this retailer.) It also provides a `t_UserId` with the id of the invited user. The consumer domain can check that the invitation is for an 'end-user' that is known to this domain. (For more details, see section on "Invitations and Announcements".)

A `t_InvitationReply` is returned which allows the consumer to inform the retailer of the action they will take regarding the invitation. (For more details, see "Invitations and Announcements".)

8.5.1.3 cancelInviteUserOutsideAccessSession()

```
void cancelInviteUserOutsideAccessSession (
      in t_ProviderId providerId,
      in SPFEEAccessCommonTypes::t_InvitationId id
) raises (
      SPFEEAccessCommonTypes::e_InvitationError
);
```

Draft definition: This operation is draft only.

Specifically, the `t_SessionInvitation` and `t_InvitationReply` parameters are defined according to the Internet Engineering Task Force working group, Multimedia Multiparty Session Control (MMUSIC) draft standard 'Session Initiation Protocol'.

This operation allows a retailer to cancel an invitation to join a service session which has been sent to a consumer. The operation can be used to cancel invitations which have been sent both within an access session, (using `inviteUser`() on `i_ConsumerInvite`), and outside of an access session, (using `inviteUserOutsideAccessSession` on `i_ConsumerInitial`).

`t_ProviderId` identifies the retailer to the consumer.

`t_InvitationId` is used in together with the `t_ProviderId` in order to determine the invitation to be cancelled. (`t_InvitationId`'s are unique to a retailer only. If a consumer has received invitations from several retailers, then invitations from different retailers may have the same id.)

If the `t_InvitationId` list is unknown to the consumer, then the operation should raise an `e_InvitationError` exception with the `InvalidInvitationId` error code.

8.5.2 i_ConsumerAccess Interface

```
// module SPFEERetConsumerAccess
interface i_ConsumerAccess : SPFEEUserAccess::i_UserAccess
{
};
```

This interface allows the retailer access to the consumer domain, during an access session. It provides operations for the retailer to request references to interfaces supported by the consumer

domain. These interfaces include those defined by Ret RP, as well as other retailer specific interfaces.

It is similar in purpose to `i_RetailerAccess`, in that it is available during the access session. It is passed to the retailer domain as part of `setUserCtxt`() on `i_RetailerNamedAccess` interface.

All the operations described below are inherited into this interface from the `i_UserAccess` interface, which supports the generic user-provider roles. No Ret RP specific specializations are defined for this interface.

The following operation signatures are taken from the module `SPFEEUserAccess`. All unscoped types need to scoped by `SPFEEUserAccess::` when used by clients of the `i_ConsumerAccess` interface.

8.5.2.1 cancelAccessSession()

```
void cancelAccessSession(
       in t_CancelAccessSessionProperties options
);
```

Draft definition: This operation is draft only.

`cancelAccessSession`() allows the retailer to end an access session with the consumer. The retailer can use this operation to terminate an access session without the consumer's permission.

When this operation is invoked, the secure and trusted relationship between the consumer and retailer has ended. Neither retailer nor consumer side interfaces available during the access session can be used to make requests. (Interfaces which have been registered for use outside an access session can still be used.)

`options` is a property list describing retailer specific options or action taken by the retailer when cancelling the access session, (i.e. the retailer may have suspended the consumer's participation in their active service sessions). Currently no specific property names and values have been defined for `t_CancelAccessSessionProperties`, and so its use is retailer specific.

This operation does not affect any contractual relationship between the consumer and retailer. The consumer can still request the establishment of an access session, and other access sessions will not have been terminated.

8.5.2.2 getInterfaceTypes()

```
void getInterfaceTypes (
       out SPFEECommonTypes::t_InterfaceTypeList itfTypes
) raises (
       SPFEECommonTypes::e_ListError
);
```

This operation returns a list of the interface types supported by the consumer domain.

`itfTypes` are all the interface types supported by the consumer domain. It is a `sequence` of `t_InterfaceTypeName`'s, which are strings representing the interface types supported by the consumer. `itfTypes` should include all the interface types that can be supported by the consumer.

If the `itfTypes` list is unavailable, because the interface types supported by the session are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.5.2.3 getInterface()

```
          void getInterface (
                in SPFEECommonTypes::t_InterfaceTypeName itfType,
                in SPFEECommonTypes::t_MatchProperties desiredProperties,
                out SPFEECommonTypes::t_InterfaceStruct itf
          ) raises (
                SPFEECommonTypes::e_InterfacesError,
                SPFEECommonTypes::e_PropertyError
);
```

This operation returns an interface, of the type requested, supported by the consumer domain.

`type` identifies the interface type of the interface reference to be returned.

The `desiredProperties` parameter can be used to identify the interface to be returned. `t_MatchProperties` identifies the properties which the sessions must match. It also defines whether a session must match one, all or none of the properties. Currently, no interface property names and values have been defined for Ret RP, and its use is retailer specific.

`itf` is returned by this operation. It contains the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of the interface type requested.

If the consumer does not support interfaces of `type`, then the operation should raise the `e_InterfacesError`, with the `InvalidInterfaceType` error code.

If an invalid property is passed, the operation should raise a `e_PropertyError`.

8.5.2.4 getInterfaces()

```
          void getInterfaces (
                out SPFEECommonTypes::t_InterfaceList itfs
          ) raises (
                SPFEECommonTypes::e_ListError
          );
```

This operation returns a list of all the interfaces supported by the consumer.

`itfs` is returned by this operation. It is a `sequence` of `t_InterfaceStruct` structures which contain the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of each interface.

If the operation cannot, or refuses to, return the interfaces, it should raise the `e_ListError` exception.

8.5.3 i_ConsumerInvite Interface

```
      // module SPFEERetConsumerAccess
      interface i_ConsumerInvite
            : SPFEEUserAccess::i_UserInvite
      {
      };
```

This interface allows the retailer to send an invitation to the consumer requesting that they join a service session. It can only be used during an access session to receive invitations. It is passed to the retailer domain as part of `setUserCtxt`() on `i_RetailerNamedAccess` interface. If the consumer wishes to receive invitations outside of an access session, then they must register the i_ConsumerInitial interface.

The operations described in the following subclauses are all inherited into this interface from the `i_UserInvite` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module SPFEEUserAccess. All unscoped types need to scoped by SPFEEUserAccess:: when used by clients of the i_ConsumerInvite interface.

8.5.3.1 inviteUser()

```
void inviteUser (
      in  SPFEEAccessCommonTypes::t_SessionInvitation invitation,
      out SPFEECommonTypes::t_InvitationReply reply
) raises (
      SPFEEAccessCommonTypes::e_InvitationError
);
```

Draft definition: This operation is draft only. Specifically, the t_SessionInvitation and t_InvitationReply parameters are defined according to the Internet Engineering Task Force working group Multimedia Multiparty Session Control (MMUSIC) draft standard 'Session Initiation Protocol'.

This operation allows a retailer to invite the consumer to join a service session. It can only be used during an access session.

t_SessionInvitation describes the service session to which the consumer has been invited, and provides an t_InvitationId to identify this invitation when joining. (It does not give interface references to the session, nor any information which would allow the consumer to join the session outside of an access session with this retailer.)

An t_InvitationReply is returned which allows the consumer to inform the retailer of the action they will take regarding the invitation. (For more details, see "Invitations and Announcements".)

The consumer may join the service session described by the invitation, from within this access session, or they may establish another access session with this retailer. The same t_InvitationId will refer to this invitation in both access sessions. The consumer should use joinSessionWithInvitation() on the i_RetailerNamedAccess interface. The service session cannot be joined without an access session with the retailer.

8.5.3.2 cancelInviteUser()

```
void cancelInviteUser (
      in SPFEECommonTypes::t_UserId               inviteeId,
      in SPFEEAccessCommonTypes::t_InvitationId   id
) raises (
      SPFEEAccessCommonTypes::e_InvitationError
);
```

Draft definition: This operation is draft only. Specifically, the t_SessionInvitation and t_InvitationReply parameters are defined according to the Internet Engineering Task Force working group Multimedia Multiparty Session Control (MMUSIC) draft standard 'Session Initiation Protocol'.

This operation allows a retailer to cancel an invitation to join a service session which has been sent to a consumer. The operation can be used to cancel invitations which have been sent both within an access session, (using inviteUser() on i_ConsumerInvite), and outside of an access session, (using inviteUserOutsideAccessSession on i_ConsumerInitial).

t_InvitationId is used in together with the t_ProviderId in order to determine the invitation to be cancelled. (t_InvitationId's are unique across all access sessions with the same retailer).

If the t_InvitationId list is unknown to the consumer, then the operation should raise an e_InvitationError exception with the InvalidInvitationId error code. (It is possible to receive

a cancelInviteUser before a corresponding inviteUser, especially if the cancel is sent just after the access session is established. This operation should raise the exception anyway.)

8.5.4 i_ConsumerTerminal Interface

```
// module SPFEERetConsumerAccess
interface i_ConsumerTerminal
       : SPFEEUserAccess::i_UserTerminal
{
};
```

This interface allows the retailer to gain information about the consumer domain's terminal configuration, and applications. It is passed to the retailer domain as part of setUserCtxt() on i_RetailerNamedAccess interface. If the consumer wishes to allow the retailer access to terminal information outside of an access session, then they must register this interface, using registerInterfaceOutsideAccessSession() on i_RetailerNamedAccess.

Draft definition: This interface is draft only. Specifically, we may enhance this interface to allow a retailer to ask more specific questions about the consumer domain.

The operations described in the following sections are all inherited into this interface from the i_UserTerminal interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module SPFEEUserAccess. All unscoped types need to scoped by SPFEEUserAccess:: when used by clients of the i_ConsumerTerminal interface.

8.5.4.1 getTerminalInfo()

```
void getTerminalInfo(
       out SPFEEAccessCommonTypes::t_TerminalInfo terminalInfo
);
```

Draft definition: This operation is draft only.

This operation allows the retailer to receive all the information about the consumer domain's terminal configuration, that the consumer wishes the retailer to have access to.

The operation returns the t_TerminalInfo structure, giving details on the type of terminal, operating system, etc. See 8.3.3, on "User Context Information".

8.5.5 i_ConsumerAccessSessionInfo Interface

```
// module SPFEERetConsumerAccess
interface i_ConsumerAccessSessionInfo
       : SPFEEUserAccess::i_UserAccessSessionInfo
{
};
```

This interface allows the retailer to inform the consumer of changes of state in other access sessions with the consumer, (e.g. access sessions with the same consumer which are created or deleted). The consumer is only informed about access sessions which they are involved in.

This interface is NOT automatically passed to the retailer, as part of setUserCtxt() on i_RetailerNamedAccess interface. If the consumer wishes to be informed of changes in other access session, then they must register this interface, using registerInterface() on i_RetailerNamedAccess. Then the retailer will tell the consumer about access session changes, until this interface is unregistered, or the current access session ends.

If the consumer wishes to be informed of access session changes outside of an access session, then they must register this interface, using registerInterfaceOutsideAccessSession() on

i_RetailerNamedAccess. The operations do not include a t_ProviderId, so if this interface is registered for use outside an access session, a separate interface must be registered with each retailer. Retailers can not share this interface, because t_AccessSessionId is only unique within a retailer for this consumer.

The operations described in the following sections are all inherited into this interface from the i_UserAccessSessionInfo interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module SPFEEUserAccess. All unscoped types need to scoped by SPFEEUserAccess:: when used by clients of the i_ConsumerAccessSessionInfo interface.

8.5.5.1 newAccessSessionInfo()

```
oneway void newAccessSessionInfo (
       in SPFEEAccessCommonTypes::t_AccessSessionInfo accessSession
   );
```

This operation is used to inform the consumer that a new access session has been established.

t_AccessSessionInfo contains the t_AccessSessionId of the new access session, the t_UserCtxtName so the consumer can tell which consumer domain/terminal the access session has been established, and t_AccessSessionProperties which are a retailer specific property list that can be used to provide more information on the access session.

8.5.5.2 endAccessSessionInfo()

```
oneway void endAccessSessionInfo (
       in SPFEEAccessCommonTypes::t_AccessSessionId asId
   );
```

This operation is used to inform the consumer that an access session has ended.

t_AccessSessionId identifies which access session has ended.

8.5.5.3 cancelAccessSessionInfo()

```
oneway void cancelAccessSessionInfo (
       in SPFEEAccessCommonTypes::t_AccessSessionId asId
   );
```

This operation is used to inform the consumer that an access session has been cancelled by the retailer. See 8.5.2.1, "cancelAccessSession()" for details. t_AccessSessionId identifies which access session has been cancelled.

8.5.5.4 newSubscribedServicesInfo()

```
oneway void newSubscribedServicesInfo (
       in SPFEEAccessCommonTypes::t_ServiceList services
   );
```

This operation is used to inform the consumer that they have been subscribed to some new services. (The consumer may have subscribed to the services through a service in this, or another access session, or a consumer may have subscribed his users to a new service.)

t_ServiceList is a list of the services that the user has subscribed to. (It is a sequence of t_ServiceInfo structures, see "Service and Session Information".)

8.5.6 i_ConsumerSessionInfo Interface

```
// module SPFEERetConsumerAccess
interface i_ConsumerSessionInfo

        : SPFEEUserAccess::i_UserSessionInfo
{
};
```

This interface allows the retailer to inform the consumer of changes of state in service sessions which the consumer is involved in. Information operations are invoked whenever a change to the service session affects the consumer, (i.e. the session is suspended), but not when the change does not affect the consumer, (i.e. another party in the session leaves). This interface is informed of changes in all service sessions involving the consumer, and not just those associated with this access session.

This interface can be passed to the retailer, as part of `setUserCtxt`() on `i_RetailerNamedAccess` interface. If the consumer does NOT wish to be informed of changes in their service sessions, then this interface does NOT need to be passed in `setUserCtxt`(). (If it is not passed, the consumer can still register this interface, using `registerInterface`() on `i_RetailerNamedAccess`. Then the retailer will tell the consumer about service session changes, until this interface is unregistered, or the current access session ends.).

If the consumer wishes to be informed of service session changes outside of an access session, then they must register this interface, using `registerInterfaceOutsideAccessSession`() on `i_RetailerNamedAccess`. The operations do not include a `t_ProviderId`, so if this interface is registered for use outside an access session, a separate interface must be registered with each retailer. Retailers can not share this interface, because `t_SessionId` is only unique within a retailer for this consumer.

The operations described in the following subclauses are all inherited into this interface from the `i_UserSessionInfo` interface, which supports the generic user-provider roles. No Ret-RP specific specialisations are defined for this interface.

The following operations are invoked when an action concerning this consumer is performed by the service session. These operations help update the access session knowledge of the involvement of the consumer in service sessions. They relate to events which eventually are usage specific (service specific), but are considered generic enough to be propagated usefully to the access session.

Only actions associated with this consumer produce info operations, i.e. consumer A receives a `endMyParticipationInfo`() invocation if they end their participation in a session, but do not receive any info if another consumer B ends their own participation. If B were to end A's participation, then A would receive the info.

All `i_ConsumerSessionInfo` interfaces receive info invocations when an action in a service session occurs. Usually one of these interfaces will be registered through each access session. It does not matter in which access session the service session is being used, all `i_ConsumerSessionInfo` interfaces will receive an info invocation.

The following operation signatures are taken from the module `SPFEEUserAccess`. All unscoped types need to be scoped by `SPFEEUserAccess::` when used by clients of the `i_ConsumerSessionInfo` interface.

```
oneway void newSessionInfo (
        in SPFEEAccessCommonTypes::t_SessionInfo session
    );
```

– The consumer has started a new service session. `session` contains information about the new session that has been started.

```
oneway void endSessionInfo (
       in SPFEECommonTypes::t_SessionId sessionId
);
```

– A service session has been ended. `sessionId` identifies the ended session.

```
oneway void endMyParticipationInfo (
       in SPFEECommonTypes::t_SessionId sessionId
);
```

– The consumer's participation in a service session has been ended. `sessionId` identifies the session.

```
oneway void suspendSessionInfo (
       in SPFEECommonTypes::t_SessionId sessionId
   );
```

– A service session has been suspended. `sessionId` identifies the session.

```
oneway void suspendMyParticipationInfo (
       in SPFEECommonTypes::t_SessionId sessionId
   );
```

– The consumer's participation in service session has been suspended. `sessionId` identifies the session.

```
oneway void resumeSessionInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );
```

– A suspended service session has been resumed. `sessionId` identifies the session. (The consumer may or may not have re-joined the service session, depending on whether they or another consumer resumed the session.) `session` contains information about the session in which has been resumed.

```
oneway void resumeMyParticipationInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );
```

– The consumer's participation in service session has been resumed. `session` contains information about the session in which the consumer has resumed their participation.

```
oneway void joinSessionInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );
```

– The consumer has joined a service session. `session` contains information about the session that the consumer has joined.

8.6 Access Interface Definitions: Retailer Domain Interfaces

This subclause describes in detail all the retailer interfaces supported over Ret RP. Each interface and all its supported operations are defined.

Many of the operations are inherited from other interfaces. However, they are described here as though they were defined on the Ret RP specified interfaces. Only the interfaces defined here must be supported for the Ret RP. It is not necessary to support the same inheritance hierarchy as defined previously in 8.2.

The following are the interfaces defined for the retailer domain of the Ret RP.

8.6.1 i_RetailerInitial Interface

```
// module SPFEERetRetailerInitial
interface i_RetailerInitial:
      SPFEEProviderInitial::i_ProviderInitial
{
         // Inherited operations shown in following subsections.
};
```

The i_RetailerInitial interface is a consumer's initial contact point with the retailer. It allows the consumer to request an access session is established between himself and the retailer.

This interface is returned when the consumer contacts the retailer. Ret-RP does not specify how the consumer contacts the retailer. Some examples could be: through the DPE naming service; through another type of directory service, such as a trader; through the SPFEE Broker business domain and Bkr reference point; or through a URL and retailer home page. An interface of this type is returned to the consumer as part of this contact the retailer scenario.

This interface inherits from i_ProviderInitial interface. It defines all of the operations which are generic to access user-provider roles, and can be re-used in other inter-domain reference points.

This interface has a role in security, and may use DPE security for message encryption, and domain authentication. That is message passing through the DPE is protected through encryption to varying, agreed levels and that both domain's credentials are exchanged for authentication. However, it does not mandate that authentication and credential acquisition occurs through the DPE, and so provides the i_RetailerAuthenticate interface to allow authentication of the user, outside of DPE security. A reference to the i_RetailerAuthenticate interface is passed to the consumer domain by the requestNamedAccess() and requestAnonymousAccess() operations if the user is not authenicated by DPE security.

The following operation signatures are taken from the module `SPFEEProviderInitial`. All unscoped types need to be scoped by `SPFEEProviderInitial::` when used by clients of the `i_RetailerInitial` interface.

8.6.1.1 requestNamedAccess()

```
void requestNamedAccess (
      in SPFEECommonTypes::t_UserId userId,
      in SPFEECommonTypes::t_UserProperties userProperties,
      out Object  namedAccessIR,                  // type:
i_ProviderNamedAccess
      out SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out SPFEEAccessCommonTypes::t_AccessSessionId asId
) raises (
      e_AccessNotPossible,
      e_AuthenticationError,
      SPFEEAccessCommonTypes::e_UserPropertiesError
);
```

The requestNamedAccess() allows the consumer to identify himself and request the establishment of an access session with the retailer. The access session provides access to use his subscribed services, etc., through a i_RetailerNamedAccess interface.

If CORBA security services are being used by both the consumer and retailer domains' DPEs, then both domain's credentials and other authentication information will be exchanged through the DPE before this operation is invoked on the retailer. This means a secure context for messages may have already been set-up between the domains, and the identity of the consumer will have been authenticated. In this case, an access session is established, and a reference to the i_RetailerNamedAccess interface will be returned. Along with this, an t_AccessSessionSecretId is returned, to be used in all requests on the new interface.

If CORBA security services are not being used, then no secure context for messages will have been set-up, and DPE messages could potentially be intercepted and read by third-parties.

If the consumer has not already been authenticated, and the DPE is unable to perform the authentication and establish an access session when this operation is invoked, then the operation will fail. An e_AuthenticationError exception will be raised, which contains a reference to a i_RetailerAuthenticate interface. This interface may be used to authenticate and set-up the secure context. Then this operation can be invoked again to establish the access session.

`userId` identifies the consumer to the retailer. For details on the structure of the `userId`, see "User Information".

`userProperties` are a sequence of user properties associated with this consumer. In general the consumer would not send sensitive information to the retailer until an access session has been established. However, this parameter can be used to pass the consumer's password to the retailer, if both domains use DPE security to encrypt the messages. Security context, and other information which is understood by the specific retailer, can also be sent. For more details, see "User Information".

If the request is successful, and the consumer has been authenticated, then the following out parameters are returned:

`namedAccessIR` is the reference to the i_RetailerNamedAccess interface, which the consumer domain uses during the access session.

NOTE – Although the IDL specifies the type (in text) as `i_ProviderNamedAccess`, it is only to state the base reference type. An abstract interface (reference) is never exported over a reference point. The `requestNamedAccess()` operation is defined inside the `i_ProviderNamedAccess` interface which is later inherited into `i_RetailerNamedAccess` and thus uses interface (references) of type `i_Retailer<...>`. The reason for stating the base reference type in the IDL is to allow re-use in other reference point definitions. For that matter, the `namedAccessIR` could just as well be of i_3ptyNamedAccess type.

`asSecretId` is an `t_AccessSessionSecretId` used whenever the consumer domain invokes an operation on the `namedAccessIR` within this access session. The `asSecretId` identifies the consumer domain from which invocations on `namedAccessIR` are made. This parameter should be used during this access session only, and only by the consumer domain to which it was returned. See 8.4.

`asId` is an `t_AccessSessionId` used to identify this access session. It is available to all the access sessions for this consumer. It can be used identity this access session when making requests on any i_RetailerNamedAccess interface between this consumer and retailer, e.g. using listServiceSessions(), an `t_AccessSessionId` can be used to scope the list to those started from a specific access session.

If the request is unsuccessful, either the consumer has not been authenticated, or the authentication has failed.

An `e_AccessNotPossible` exception is raised if the retailer is unable, or refuses to allow the consumer domain to establish an access session with them.

An `e_AuthenticationError` exception is raised if the retailer has not authenticated the consumer. This contains a list of authentication methods that can be used with the i_RetailerAuthenticate interface. The interface is returned as either an interface reference, or a stringified object reference, depending on the retailer. This reference is used to authenticate the consumer with the retailer. Once the consumer has been successfully authenticated, (using one of the authentication methods indicated), then the consumer can call this operation again to request the establishment of an access session, and get a reference to the i_RetailerNamedAccess interface.

If an e_UserPropertiesError exception is raised, then there is a problem with the userProperties. The errorCode provides the reason for the error.

8.6.1.2 requestAnonymousAccess()

```
void requestAnonymousAccess (
       in SPFEECommonTypes::t_UserProperties userProperties,
       out Object anonAccessIR,       // type: i_ProviderAnonAccess
       out SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       out SPFEEAccessCommonTypes::t_AccessSessionId asId

) raises (
       e_AccessNotPossible,
       e_AuthenticationError
       SPFEEAccessCommonTypes::e_UserPropertiesError
);
```

The requestAnonymousAccess() allows the consumer to request the establishment of an access session with the retailer. It is used when the consumer does not have a user identity with the retailer. This may be because they have not previously contacted this retailer, or they wish to remain anonymous to this retailer.

This operation returns a reference to a i_RetailerAnonAccess interface, through which the consumer can access services, and register as a named user with the retailer, if they wish to do so.

If CORBA security services are being used by both the consumer and retailer domains' DPEs, then both domain's may exchange credentials through the DPE before this operation is invoked on the retailer. This means a secure context for messages may have already been set-up between the domains, but the credentials will not contain any information about the identity of the specific consumer.

If CORBA security services are not being used, then no secure context for messages will have been set-up, and DPE messages could potentially be intercepted and read by third-parties.

userProperties are a sequence of user properties associated with this consumer. They may contain security context and other information which is understood by the specific retailer. For more details, see "User Information".

If the request is successful, an access session has been established with the consumer. The following out parameters are returned:

anonAccessIR is the reference to the i_RetailerAnonAccess interface, which the consumer domain uses during the access session.

asSecretId is an t_AccessSessionSecretId used whenever the consumer domain invokes an operation on the namedAccessIR within this access session. The asSecretId identifies the consumer domain from which invocations on namedAccessIR are made. This parameter should be used during this access session only, and only by the consumer domain to which it was returned. See 8.4.

asId is an t_AccessSessionId used to identify this access session. It is available to all the access sessions for this consumer. It can be used identity this access session when making requests on any i_RetailerNamedAccess interface in an access session between this consumer and retailer. (In general, anonymous users can only have one access session with the retailer, as each access session with each anonymous user must be treated separately. Since the consumers are anonymous to the retailer, each consumer appears to be a separate individual, even if they are, in fact, the same person.)

If the request is unsuccessful, either the consumer has not been authenticated, or the authentication has failed.

An e_AccessNotPossible exception is raised if the retailer is unable, or refuses to allow the consumer domain to establish an access session with them.

An e_AuthenticationError exception is raised if the retailer requires that the consumer domain is authenticated. This contains a list of authentication methods that can be used with the i_RetailerAuthenticate interface. (Authentication methods may authenticate the domains only, and not the specific consumer.) The interface is returned as either an interface reference, or a stringified object reference, depending on the retailer. This reference is used to authenticate the consumer domain with the retailer. Once the consumer domain has been successfully authenticated, (using one of the authentication methods indicated), then the consumer can call this operation again to request the establishment of an access session, and get a reference to the i_RetailerAnonAccess interface.

If an e_UserPropertiesError exception is raised, then there is a problem with the userProperties. The errorCode provides the reason for the error.

8.6.2 i_RetailerAuthenticate Interface

```
// module SPFEERetRetailerInitial
interface i_RetailerAuthenticate:
       SPFEEProviderInitial::i_ProviderAuthenticate
{
           // Inherited operations shown in following subsections.
};
```

The i_RetailerAuthenticate interface allows the consumer and the retailer to be authenticated. It provides a generic mechanism for authentication which can be used to support a number of different authentication protocols.

The purpose of this interface is to verify to the consumer and retailer that each domain is interacting with the domain they have been told they are talking to. This means mutual authentication of both domains. Other authentication schemes which authenticate only one of the domains is also possible using this interface. The interface provides a set of generic operations, that can be used in authentication. However, the operations only provide a mechanism for 'transporting' authentication information. Both domains must know and use a common authentication protocol, and perform this protocol using these operations in order to authenticate the domains. Ret-RP does not specify any particular authentication protocol. The getAuthenticationMethods() operation on this interface can be used to determine the authentication protocols supported by the retailer, and a protocol chosen for authentication. The authentication protocol may, or may not identify the individual consumer. It may only identy and authenticate the consumer's domain.

The following operation signatures are taken from the module SPFEEProviderInitial. All unscoped types need to be scoped by SPFEEProviderInitial:: when used by clients of the i_RetailerInitial interface.

8.6.2.1 getAuthenticationMethods()

```
void getAuthenticationMethods {
       in  t_AuthMethodSearchProperties desiredProperties,
       out t_AuthMethodDescList authMethods
) raises (
       e_AuthMethodPropertiesError,
       SPFEECommonTypes::e_ListError
);
```

The getAuthenticationMethods() allows the consumer to ask the retailer for a list of the authentication methods supported. A particular authentication method can then be chosen by the consumer to use in authenticate().

desiredProperties is a list containing the properties that the consumer wishes the authentication method to support. (See t_MatchProperties in "Properties and Property Lists"). For example, the

consumer can request that the authentication methods returned support mutual authentication, or retailer authentication only. Currently no specific property names and values have been defined for `t_AuthMethodSearchProperties`, and so its use is retailer specific.

`authMethods` is a list of authentication methods which match the `desiredProperties`, and which the retailer supports. The `t_AuthMethodDesc` structure contains the authentication method identifier, and a list of properties of the method. It is assumed that both the consumer and retailer both know the protocol to follow in order to use the authentication method defined.

The `authMethods` list may be empty. This may occur if the retailer does not support any methods matching the properties requested, or if the retailer does not wish to allow the consumer to authenticate using a method with the desired properties. e.g. if the consumer requests a method for retailer only authentication, and the retailer wishes to have mutual authentication.

If the `desiredProperties` parameter is wrongly formatted, or provides an invalid property name or value, the `e_PropertyError` exception should be raised. (Property names which are not recognised can be ignored, if `desiredProperties` requires that only some, or none of the properties are matched.)

If the `authMethods` list is unavailable, then raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.2.2 authenticate()

```
void authenticate(
      in t_AuthMethod authMethod,
      in string securityName,
      in t_opaque authenData,
      in t_opaque privAttribReq,
      out t_opaque privAttrib,
      out t_opaque continuationData,
      out t_opaque authSpecificData,
      out t_AuthenticationStatus authStatus
) raises (
      e_AuthMethodNotSupported
);
```

`authenticate`() allows the consumer to select an authentication method, pass authentication data to the retailer.

Once the consumer domain has determined an authentication method with the retailer, this operation is used to transport authentication data, and other credentials to the retailer. This data is used to perform the type of authentication appropriate to the authentication method, (this may be mutual authentication, or authentication of the consumer/retailer domain only, etc.).

The retailer then returns its authentication data (if required), challenge data for the consumer to respond using continueAuthentication() (if required), and the requested credentials (if possible). If further authentication protocol is required before credentials are returned then these can be returned by continueAuthentication().

The following parameters are sent by the consumer to the retailer:

`authMethod` is used to identify the authentication method proposed by the consumer. It affects the composition and generation method of the other opaque data parameters. Currently no specific authentication methods values have been defined for `t_AuthMethod`, and so its use is retailer specific.

`securityName` is the name assumed by consumer for authentication. It may be an empty string according to the authentication method used.

`authenData` is opaque data containing consumer attributes to be authenticated. Its format depends upon the authentication method used.

`privAttribReq` is opaque data which is used to specify the rights and privileges which the consumer domain requests from the retailer domain. This data may correspond to levels of security to access different areas of the retailer domain. Its format depends upon the authentication method used.

The following parameters are returned by the retailer to the consumer:

`privAttrib` is opaque data which defines the privilege attributes granted to the consumer, based upon the `privAttribReq`, and their authentication data. Its format depends upon the authentication method used.

`continuationData` is opaque data which is used to challenge the consumer. The consumer has not yet been authenticated, and must process this data and return the result to the retailer using the `continueAuthentication()` operation. Its format depends upon the authentication method used. This parameter may be ignored if the value of `authStatus` is not `SecAuthContinue`.

`authSpecificData` is opaque data which is specific to the authentication method used.

`authStatus` identifies the status of the authentication process. It is an enumerated type with the following values:

– SecAuthSuccess:

  Authentication has completed successfully. No (more) calls to `continueAuthentication()` are necessary. The consumer can call requestNamedAccess() on i_RetailerInitial interface to gain a reference to the i_RetailerNamedAccess interface. (Or if they wish to be an anonymous user, call requestAnonymousAccess() for a i_RetailerAnonAccessinterface.).

– SecAuthFailure:

  Authentication has completed unsuccessfully. The consumer has not been authenticated, and will not be able to establish an access session. Calls to requestNamedAccess() will continue to raise an `e_AccessNotPossible`, or `e_AuthenticationError` exception.

– SecAuthContinue:

  Authentication is continuing, and the consumer must reply to this result by calling `continueAuthentication()`.

– SecAuthExpired:

  Authentication has timed out. The consumer did not make this invocation of continueAuthentication() quickly enough, after the reply from authenticate(), or the previous call to continueAuthentication(). Authentication must be started again from the beginning by calling authenticate(). This enumeration should not be returned by authenticate().

8.6.2.3 continueAuthentication()

```
void continueAuthentication{
      in t_opaque responseData,
      out t_opaque privAttrib,
      out t_opaque continuationData,
      out t_opaque authSpecificData,
      out t_AuthenticationStatus authStatus
);
```

`continueAuthentication`() allows the consumer to continue an authentication protocol, started using `authenticate`(), and pass authentication data to the retailer.

This operation should be invoked by the consumer if the `authStatus` returned from `authenticate`(), or a previous call to `continueAuthentication`(), is `SecAuthContinue`. The `authStatus` is used by both operations to indicate if the consumer needs to make another call to this

operation. Parameters returned by this operation must be processed by the consumer according to the authentication method, and the results provided as in parameters to the subsequent call to this operation.

`responseData` is opaque data from the consumer. This data has been generated by the consumer according to the authentication method, based on the `continuationData` returned by the previous call to `authenticate`() or `continueAuthentication`(). Precisely how this data is generated, and formatted is specific to the authentication method used.

`continuationData` is opaque data which is used to challenge the consumer. The consumer has not yet been authenticated, and must process this data and return the result to the retailer using the `continueAuthentication()` operation. Its format depends upon the authentication method used. This parameter may be ignored if the value of `authStatus` is not `SecAuthContinue`.

`authSpecificData` is opaque data which is specific to the authentication method used.

`authStatus` identifies the status of the authentication process. It has the same values as for `authenticate`().

8.6.3 i_RetailerAccess Interface

```
// module SPFEEProviderAccess
interface i_RetailerAccess
{
};
```

i_RetailerAccess interface is an abstract interface, used to inherit common operations in to the i_RetailerNamedAccess, and i_RetailerAnonAccess interfaces.

The purpose of this interface is for inheritance, as described above. It should not be available over the Ret-RP. No instances of this interface type should be created.

Currently no operations are defined for this interface. It will be contain operations which are shared between the i_RetailerNamedAccess, and i_RetailerAnonAccess interfaces. Currently all operations are defined on the i_RetailerNamedAccess interface, and no operations have been identified for the i_RetailerAnonAccess interface.

8.6.4 i_RetailerNamedAccess Interface

```
// module SPFEEProviderAccess
interface i_RetailerNamedAccess
       : i_ProviderNamedAccess, i_RetailerAccess
{
          // Inherited operations shown in following subsections.
};
```

i_RetailerNamedAccess interface allows a known consumer access to his subscribed services. The consumer uses it for all operations within an access session with the retailer.

This interface is returned when the consumer has been authenticated by the retailer and an access session has been established. It is returned by calling requestNamedAccess() on the i_RetailerInitial interface.

This interface inherits from i_ProviderNamedAccess and i_RetailerAccess interfaces. i_ProviderNamedAccess defines all of the operations which are generic to access user-provider roles, and can be re-used in other inter-domain reference points. All the operations on this interface are inherited from there. The i_RetailerAccess interface is currently blank. It will contain operations which are shared between the i_RetailerAnonAccess interface, and this interface, that are specific to the Ret RP.

The following operation signatures are taken from the module SPFEEProviderAccess. All unscoped types need to scoped by SPFEEProviderAccess:: when used by clients of the i_RetailerAccess interface.

8.6.4.1 setUserCtxt()

```
void setUserCtxt (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in t_UserCtxt userCtxt
) raises (
       SPFEEAccessCommonTypes::e_AccessError,

       e_UserCtxtError
);
```

The setUserCtxt() allows the consumer to inform the retailer about interfaces in the consumer domain, and other consumer domain information, (e.g. user applications available in the consumer domain, operating system used, etc).

userCtxt is a structure containing consumer domain configuration information and interfaces.

This operation should be called immediately after receiving the reference to this interface. If this operation has not been called successfully, subsequent operations may raise an e_AccessError exception with a UserCtxtNotSet error code.

If there is a problem with userCtxt, then e_UserCtxtError should be raised with the appropriate error code.

8.6.4.2 getUserCtxt()

```
void getUserCtxt (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_UserCtxtName ctxtName,
       out t_UserCtxt userCtxt
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_UserCtxtError
);
```

This operation allows the consumer to retrieve information about user contexts that have been registered with the retailer.

ctxtName is the name of the context that the consumer wishes to retrieve user context information about. (ctxtName is set by the consumer when registering a user context, and is the consumer term for the context, e.g. "Home", "Work", "Mum's House", etc.

userCtxt is a structure containing consumer domain configuration information and interfaces.

8.6.4.3 getUserCtxts()

```
void getUserCtxts (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in t_SpecifiedUserCtxt ctxt,
       out t_UserCtxtList userCtxts
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_UserCtxtError,
       SPFEECommonTypes::e_ListError
);
```

8.6.4.4 listAccessSessions()

```
void listAccessSessions (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out SPFEEAccessCommonTypes::t_AccessSessionList asList
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEECommonTypes::e_ListError
);
```

The listAccessSessions() returns a list of access sessions. The list contains all the access sessions the consumer currently established with this retailer. It is a `sequence` of `t_AccessSessionInfo` structures, which consist of the `t_AccessSessionId`, `t_UserCtxtName`, and `t_AccessSessionProperties`. The last of these is a `t_PropertyList`. Currently no specific property names and values have been defined for `t_AccessSessionProperties`, and so its use is retailer specific.

The information returned by this operation can be used by the consumer to find out which other access sessions are currently established; end some of those access sessions (see `endAccessSession()`); list the service sessions of those access sessions (see `listServiceSessions()`); and be informed of changes to those access sessions and service sessions (see `i_ConsumerAccessSessionInfo` and `i_ConsumerSessionInfo` interfaces).

If the `asList` list is unavailable, because the consumer's access sessions are not available, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.4.5 endAccessSessions()

```
void endAccessSession(
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as,
      in t_EndAccessSessionOption option
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
      e_EndAccessSessionError
);
```

The `endAccessSession()` allows the consumer to end an access session.

The operation can end the current access session; a specified access session; or all access sessions (including the current one), through the use of the `t_SpecifiedAccessSession` parameter.

`t_EndAccessSessionOptions` allows the consumer to choose the actions the retailer should take, if there are active or suspended service sessions, when the access session ends. The actions are only used as part of this invocation. The retailer does not remember the action chosen. (Retailers may define a default policy for service sessions when a consumer ends the access session in which they were created, or allow the consumer to define the policy. Currently, Ret-RP does not support the definition of such a policy by the consumer.)

If `as` is wrongly formatted, or provides an invalid access session id, then the `e_SpecifiedAccessSessionError` exception should be raised.

`e_EndAccessSessionError` is raised if `option` is invalid, or service sessions remain active, or suspended, which are not allowed by the retailer. (A consumer may end an access session, leaving active or suspended sessions if this is allowed as a policy of the retailer for this consumer.)

8.6.4.6 getUserInfo()

```
void getUserInfo(
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out SPFEEAccessCommonTypes::t_UserInfo userInfo
) raises (
      SPFEEAccessCommonTypes::e_AccessError
);
```

The `getUserInfo()` allows the consumer to request information about himself.

This operation returns a `t_UserInfo` structure as an out parameter. This contains the consumer's `t_UserId`, their name, and a list of user properties. Currently no specific property names and values have been defined for `t_UserProperties`, and so its use is retailer specific.

8.6.4.7 listSubscribedServices()

```
void listSubscribedServices (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_SubscribedServiceProperties desiredProperties,
      out SPFEEAccessCommonTypes::t_ServiceList services
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEECommonTypes::e_PropertyError,
      SPFEECommonTypes::e_ListError
);
```

The `listSubscribedServices()` returns a list of the services to which the consumer has previously been subscribed.

The `desiredProperties` parameter can be used to scope the list of subscribed services. `t_SubscribedServiceProperties` identifies the properties which the suscribed services must match. It also defines whether a subscribed service must match one, all or none of the properties. (See `t_MatchProperties` in "Properties and Property Lists".) Currently no specific property names and values have been defined for `t_SubscribedServiceProperties`, and so its use is retailer specific.

The list of services subscribed to by the consumer, and matching the `desiredProperties`, is returned in the `t_ServiceList`. This is a `sequence` of `t_ServiceInfo` structures, which contain the `t_ServiceId`, `t_UserServiceName` (consumers name for the service), and a `sequence` of service properties, `t_ServiceProperties`. Currently no specific property names and values have been defined for `t_ServiceProperties`, and so its use is retailer specific.

If the `desiredProperties` parameter is wrongly formatted, or provides an invalid property name or value, the `e_PropertyError` exception should be raised. (Property names which are not recognised can be ignored, if `desiredProperties` requires that only some, or none of the properties are matched.)

If the `services` list is unavailable, because the retailer's services are not available, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.4.8 discoverServices()

```
void discoverServices(
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_DiscoverServiceProperties desiredProperties,
      in unsigned long howMany,
      out SPFEEAccessCommonTypes::t_ServiceList services,
      out Object iteratorIR       // type: i_DiscoverServicesIterator
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEECommonTypes::e_PropertyError,
      SPFEECommonTypes::e_ListError
);
```

The discoverServices() returns a list of the services available from this retailer.

This operation is used to discover the services provided by the retailer, for use by the consumer. It can be used to retrieve information on all of the services provided, or be scoped by the desiredProperties parameter. (See t_MatchProperties in "Properties and Property Lists".)

The list of retailer services matching the desiredProperties is returned in services. This is a sequence of t_ServiceInfo structures, which contain the t_ServiceId, t_UserServiceName (consumers name for the service), and a sequence of service properties, t_ServiceProperties. Currently no specific property names and values have been defined for t_ServiceProperties, and so its use is retailer specific.

The howMany parameter defines the number of t_ServiceInfo structures to return in the services parameter. The length of services will not exceed this number. Any remaining services which match the desiredProperties, but which aren't included in services are accessible through iteratorIR, the i_DiscoverServicesIterator interface. If there are no remaining services, then iteratorIR should be null.

If the desiredProperties parameter is wrongly formatted, or provides an invalid property name or value, the e_PropertyError exception should be raised. (Property names which are not recognised can be ignored, if desiredProperties requires that only some, or none of the properties are matched.)

If the services list is unavailable, because the retailer's services are not available, then the operation should raise an e_ListError exception with the ListUnavailable error code.

8.6.4.9 getServiceInfo()

```
void getServiceInfo (
     in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
     in SPFEEAccessCommonTypes::t_ServiceId serviceId,
     in SPFEEProviderAccess::t_SubscribedServiceProperties
     desiredProperties,
     out SPFEEAccessCommonTypes::t_ServiceProperties serviceProperties
) raises (
     SPFEEAccessCommonTypes::e_AccessError,
     SPFEEProviderAccess::e_ServiceError
);
```

The getServiceInfo() returns information on a specific service, identified by the serviceId. The desiredProperties list can scope the information which is requested to be returned.

8.6.4.10 listRequiredServiceComponents()

```
void listRequiredServiceComponents (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEEAccessCommonTypes::t_ServiceId serviceId,
       in SPFEEAccessCommonTypes::t_TerminalConfig terminalConfig,
       in SPFEEAccessCommonTypes::t_TerminalInfo terminalInfo,
       // Example of usage for Java applet download:
       // name-value pair describing the url of a Java applet
       // name = "URL"
       // type = "string"
       out SPFEECommonTypes::t_PropertyList locations
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       SPFEEProviderAccess::e_ServiceError
);
```

This operation retrieves information on how to download the ssUAP in case of Java applets. The terminalInfo is included as an IN parameter to avoid an explicit call of the getTerminalInfo operation.

8.6.4.11 listServiceSessions()

```
void listServiceSessions (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as,
       in t_SessionSearchProperties desiredProperties,
       out SPFEEAccessCommonTypes::t_SessionList sessions
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
       SPFEECommonTypes::e_PropertyError,
       SPFEECommonTypes::e_ListError
);
```

The `listServiceSessions()` returns a list of the service sessions, which the consumer is involved in. This includes active and suspended sessions.

The `as` parameter scopes the list of sessions by the access session in which they are used. It can identify the current access session; a list of access sessions; or all access sessions. (A session is associated with an access session if it is being used within that access session, or it has been suspended (or participation suspended), and was being used within that access session when it was suspended.)

The `desiredProperties` parameter can be used to scope the list of sessions. `t_SessionSearchProperties` identifies the properties which the sessions must match. It also defines whether a session must match one, all or none of the properties. (See `t_MatchProperties` in "Properties and Property Lists"). The following property names and values have been defined for `t_SessionSearchProperties:`

– name: `"SessionState"`

  value: `t_SessionState`

  If a property in `t_SessionSearchProperties` has the name `"SessionState"`, then the session must have the same `t_SessionState` as given in the propery value.

– name: `"UserSessionState"`

  value: `t_UserSessionState`

  If a property in `t_SessionSearchProperties` has the name `"UserSessionState"`, then the session must have the same `t_UserSessionState` as given in the propery value.

Other retailer specific properties can also be defined in `desiredProperties`.

The list of sessions matching the `desiredProperties` and the `accessSession` are returned in `sessions`. This is a `sequence` of `t_SessionInfo` structures, which define the `t_SessionId`, `t_ParticipantSecretId`, `t_PartyId`, `t_UserSessionState`, `t_InterfaceList`, `t_SessionModelList`, and `t_SessionProperties` of the session.

If `as` is wrongly formatted, or provides an invalid access session id, then the `e_SpecifiedAccessSessionError` exception should be raised.

If the `desiredProperties` parameter is wrongly formatted, or provides an invalid property name or value, the `e_PropertyError` exception should be raised. (Property names which are not recognised can be ignored, if `desiredProperties` requires that only some, or none of the properties are matched.)

If the `sessions` list is unavailable, because the consumer's sessions are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.4.12 getSessionModels()

```
void getSessionModels (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_SessionId sessionId,
      out SPFEECommonTypes::t_SessionModelList sessionModels
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      e_SessionError,
      SPFEECommonTypes::e_ListError
);
```

The `getSessionModels()` returns a list of the session models supported by a service session. It can be used on active and suspended sessions.

`sessionId` identifies the session whose session models are retrieved.

`sessionModels` are the session models supported by the session. It is a `sequence` of `t_SessionModel` structures, which contain the name of the session model, and a list of properties for that session model. A model has been defined in SPFEE which is called the SPFEE Session Model. Further information on this sophisticated object oriented generic session model is available [6], but outside the scope of this Supplement.

`e_SessionError` is raised if the `sessionId` is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return `sessionModels`.

If the `sessionModels` list is unavailable, because the session models supported by the session are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.4.13 getSessionInterfaceTypes()

```
void getSessionInterfaceTypes (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_SessionId sessionId,
      out SPFEECommonTypes::t_InterfaceTypeList itfTypes
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      e_SessionError,
      SPFEECommonTypes::e_ListError
);
```

The `getSessionInterfaceTypes()` returns a list of the interface types supported by a service session. It can be used on active and suspended sessions.

`sessionId` identifies the session whose interface types are retrieved.

`itfTypes` are all the interface types supported by the session. It is a `sequence` of `t_InterfaceTypeName`'s, which are strings representing the interface types supported by the session. `itfTypes` should include all the interface types that can be supported by the session.

`e_SessionError` is raised if the `sessionId` is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return `itfTypes`.

If the `itfTypes` list is unavailable, because the interface types supported by the session are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.4.14 getSessionInterface()

```
void getSessionInterface (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId,
       in SPFEECommonTypes::t_InterfaceTypeName itfType,
       out SPFEECommonTypes::t_InterfaceStruct itf
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError,
       SPFEECommonTypes::e_InterfacesError
);
```

The `getSessionInterface()` returns an interface, of the type requested, supported by a service session. It can be used on active sessions.

`sessionId` identifies the session whose interface are retrieved.

`itfType` identifies the interface type of the interface reference to be returned.

`itf` is returned by this operation. It contains the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of the interface type requested.

`e_SessionError` is raised if the `sessionId` is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return `itfTypes`.

If the session does not support interfaces of `itfType`, then the operation should raise the `e_SessionInterfacesError`, with the `InvalidSessionInterfaceType` error code.

8.6.4.15 getSessionInterfaces()

```
void getSessionInterfaces (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId,
       out SPFEECommonTypes::t_InterfaceList itfs
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError,
       SPFEECommonTypes::e_ListError
);
```

The `getSessionInterfaces()` returns a list of all the interfaces supported by a service session. It can be used on active sessions only.

`sessionId` identifies the session whose interface types are retrieved.

`itfs` is returned by this operation. It is a `sequence` of `t_InterfaceStruct` structures which contain the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of each interface.

`e_SessionError` is raised if the `sessionId` is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return `itfTypes`.

If the `itfs` list is unavailable, because the interface supported by the session are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

8.6.4.16 listSessionInvitations()

```
void listSessionInvitations (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       out SPFEEAccessCommonTypes::t_InvitationList invitations
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       SPFEECommonTypes::e_ListError
);
```

The listSessionInvitations() returns a list of the invitations to join a service session, which have been sent to the consumer through this retailer.

invitations is returned by this operation. It is a sequence of t_SessionInvitation structures:

```
struct t_SessionInvitation {
       t_InvitationId id;
       t_UserId inviteeId;
       t_SessionPurpose purpose;
       t_InvitationReason reason;
       t_InvitationOrigin origin;
};
```

id identifies the particular invitation. It uniquely identifies this invitation from others for this consumer at this retailer. (Other consumers with this retailer may have invitations with the same id). This id is used in joinSessionWithInvitation() to join the session refered to by this invitation.

inviteeId is the user id of this consumer. (It is not necessary here, as the user id is known through the access session. It is included in this structure to allow invitations to be deliverable outside of an access session, and allow the receiptant to check that the invitation was for them.)

purpose is a string containing the purpose of the session.

reason is a string containing the reason this consumer has been invited to join this session.

origin is a structure containing the userId of the consumer that requested that the invitation was sent to this consumer, and their sessionId for the session that this consumer has been invited to join. (The sessionId is provided so that if the invited consumer contacts the inviting consumer, he is able to tell which session the invited consumer is refering to.)

If the invitation list is not available, then the operation should raise the e_ListError, with the ListUnavailable error code.

8.6.4.17 listSessionAnnouncements()

```
void listSessionAnnouncements (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in t_AnnouncementSearchProperties desiredProperties,
       out SPFEECommonTypes::t_AnnouncementList announcements
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       SPFEECommonTypes::e_PropertyError,
       SPFEECommonTypes::e_ListError
);
```

The listSessionAnnouncements() returns a list of the session announcements, which have been announced through this retailer.

Sessions can be announced due to requests from session participants (see Multiparty Feature Set), or due to properties of the session intialisation, service factory or policies of the user starting the service. The process by which sessions are announced is not defined by Ret-RP. However, this operation is provided in order to allow a consumer to request a list of sessions which have been announced. (Announcements may be scoped in order to restrict the distribution of the announcement to particular groups.) This operation returns a list of announcements which match the desiredProperties, as specified by the consumer.

The desiredProperties parameter can be used to scope the list of announcements. t_AnnouncementSearchProperties identifies the properties which the announcements must match. (See t_MatchProperties in "Properties and Property Lists"). Currently no specific property names and values have been defined for t_AnnouncementSearchProperties, and so its use is retailer specific.

announcements is a list of announcements available to the consumer, and matching the desiredProperties. This is a sequence of t_SessionAnnouncement structures, which contain the properties od the announcement, t_AnnouncementProperties. Currently no specific property names and values have been defined for t_AnnouncementProperties, and so its use is retailer specific.

If the desiredProperties parameter is wrongly formatted, or provides an invalid property name or value, the e_PropertyError exception should be raised. (Property names which are not recognised can be ignored, if desiredProperties requires that only some, or none of the properties are matched.)

If an announcement list is not available, then the operation should raise the e_ListError, with the ListUnavailable error code.

8.6.4.18 startService()

```
void startService (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEEAccessCommonTypes::t_ServiceId serviceId,
      in t_ApplicationInfo app,
      in SPFEECommonTypes::t_SessionModelReq sessionModelReq,
      in t_StartServiceUAProperties uaProperties,
      in t_StartServiceSSProperties ssProperties,
      out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      e_ServiceError,
      e_ApplicationInfoError,
      SPFEECommonTypes::e_SessionModelError,
      e_StartServiceUAPropertyError,
      e_StartServiceSSPropertyError
);
```

The startService() starts a service session for the consumer.

serviceId is the service type identifier, which indicates the service type of the session which the consumer wishes to start.

app is a structure containing information on the application, which will be used to interact with the service session. It includes: application name, version, serial number, property list, etc. It also includes: a list of interfaces supported by the application, which can optionally include references to some of those interfaces if they are available; a list of session models, and feature sets, again including interface references if appropriate; and a stream interface description list.

sessionModelReq defines the session models and feature sets that the consumer domain wishes the session to have. It allows the consumer to request that some, all or none of the session models are supported by the session.

uaProperties is a property list that will be interpreted by the retailer domain before the service session is started. No property names or values are defined, so it use it retailer-specific. Its purpose is to allow the consumer to define some preferences or other constraints that they wish to be applied to this service session only, and that the retailer needs to know before the session is started. (These properties may affect the choice of service factory for the session.)

ssProperties is a property list that will be interpreted by the service session, as soon as it has started. (i.e. before the references to the session are returned to the consumer domain). No property names or values are defined. Its use is entirely service specific, and only the service session is intended to interpret the properties given. (This parameter allows the consumer domain/application to pass service specific information to the service session, which is not intended for the retailer domain to interpret.)

sessionInfo is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See "Service and Session Information".)

The following are exceptions which are raised by this operation:

e_ServiceError is raised if the serviceId is invalid/unknown by the retailer, or if a service session cannot be created.

e_ApplicationInfoError is raised if there are unknown or invalid values for t_ApplicationInfo, or if the application is incompatable with the type of service being started.

e_SessionModelError is raised if invalid session models and/or feature sets are required for the service session.

e_StartServiceUAPropertyError is raised if there is an error in the properties for uaProperties. It has the same properties error codes as e_PropertyError. (See "Properties and Property Lists" for more details.)

e_StartServiceSSPropertyError is raised if there is an error in the properties of ssProperties. It has the same properties error codes as e_PropertyError.

8.6.4.19 endSession()

```
void endSession (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError
);
```

The endSession() ends a service session for the consumer. It can be used to end sessions which the consumer is currently active in, and sessions which have been suspended, or the consumer has suspended his participation.

sessionId is the identifier of the session to be ended.

The exception e_SessionError is raised if sessionId is invalid; or the session refuses to end because of the user's session state; or the user does not have permission.

8.6.4.20 endMyParticipation()

```
void endMyParticipation (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError
);
```

The endMyParticipation() ends the consumer's participation in a service session. It can be used on a session which the consumer is currently active in, or which has been suspended, or the consumer has suspended his participation.

sessionId is the identifier of the session to end this user's participation.

The exception e_SessionError is raised if sessionId is invalid; or the session refuses to end this user's participation because of their session state; or the user does not have permission.

8.6.4.21 suspendSession()

```
void suspendSession (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError
);
```

The `suspendSession()` suspends a service session for the consumer. It can be used to suspend sessions which the consumer is currently active in, and sessions which the consumer has already suspended his participation.

`sessionId` is the identifier of the session to suspend.

The exception `e_SessionError` is raised if `sessionId` is invalid; or the session refuses to suspend because of this user's session state; or the user does not have permission.

8.6.4.22 suspendMyParticipation()

```
void suspendMyParticipation (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError
);
```

The `suspendMyParticipation()` suspends the consumer's participation in a service session. It can be used on a session which the consumer is currently active in.

`sessionId` is the identifier of the session to suspend this user's participation.

The exception `e_SessionError` is raised if `sessionId` is invalid; or the session refuses to suspend this user's participation because of their session state; or the user does not have permission.

8.6.4.23 resumeSession()

```
void resumeSession (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId,
       in t_ApplicationInfo app,
       out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError,
       e_ApplicationInfoError
);
```

The `resumeSession()` resumes a service session. It is used on a session which is suspended.

`sessionId` is the identifier of the session to resume.

`app` is a structure containing information on the application, which will be used to interact with the service session. This application may be different to the user's original application that they were using when the session was suspended.

`sessionInfo` is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See "Service and Session Information".)

The exception `e_SessionError` is raised if `sessionId` is invalid; or the session refuses to resume because of the user's session state; or the user does not have permission.

The exception e_ApplicationInfoError is raised if there are unknown or invalid values for t_ApplicationInfo, or if the application is incompatable with the type of service being resumed.

8.6.4.24 resumeMyParticipation()

```
void resumeMyParticipation (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEECommonTypes::t_SessionId sessionId,
       in t_ApplicationInfo app,
       out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError,
       e_ApplicationInfoError
);
```

The resumeMyParticipation() resumes the consumer's participation in a service session. It can be used on a session which the consumer has previously suspended his participation from.

sessionId is the identifier of the session to resume the user's participation.

app is a structure containing information on the application, which will be used to interact with the service session. This application may be different to the user's original application that they were using when they suspended their participation.

sessionInfo is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See "Service and Session Information".)

The exception e_SessionError is raised if sessionId is invalid; or the session refuses to resume the user's participation because of their session state; or they do not have permission.

The exception e_ApplicationInfoError is raised if there are unknown or invalid values for t_ApplicationInfo, or if the application is incompatable with the type of service being resumed.

8.6.4.25 joinSessionWithInvitation()

```
void joinSessionWithInvitation (
       in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
       in SPFEEAccessCommonTypes::t_InvitationId invitationId,
       in t_ApplicationInfo app,
       in SPFEECommonTypes::t_PropertyList  joinProperties,
       out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
       SPFEEAccessCommonTypes::e_AccessError,
       e_SessionError,
       SPFEEAccessCommonTypes::e_InvitationError,
       e_ApplicationInfoError
);
```

The joinSessionWithInvitation() allows the consumer to join an existing service session, to which the consumer has received an invitation.

invitationId is the identifier of the invitation. The invitation, kept by the retailer, contains sufficient information for retailer to contact the service session, and request that the consumer be allowed to join the session.

app is a structure containing information on the application, which will be used to interact with the service session.

sessionInfo is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part

of the session, including the interface references to interact with the session. (See "Service and Session Information".)

The exception `e_SessionError` is raised if the session refuses to allow the consumer to join it.

The exception `e_InvitationError` is raised if the `invitationId` is invalid.

The exception `e_ApplicationInfoError` is raised if there are unknown or invalid values for `t_ApplicationInfo`, or if the application is incompatable with the type of service being joined.

8.6.4.26 joinSessionWithAnnouncement()

```
void joinSessionWithAnnouncement (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEEAccessCommonTypes::t_AnnouncementId announcementId,
      in t_ApplicationInfo app,
      in SPFEECommonTypes::t_PropertyList  joinProperties,
      out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      e_SessionError,
      e_AnnouncementError,
      e_ApplicationInfoError
);
```

The `joinSessionWithAnnouncement()` allows the consumer to join an existing service session, to which the consumer has discovered an announcement. Session announcements may be gained in a number of ways (not described in Ret-RP), including through a specialized service session.

`announcementId` is the identifier of the announcement. The announcement, kept by the retailer, contains sufficient information for retailer to contact the service session, and request that the consumer be allowed to join the session.

`app` is a structure containing information on the application, which will be used to interact with the service session.

`sessionInfo` is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See "Service and Session Information".)

The exception `e_SessionError` is raised if the session refuses to allow the consumer to join it.

The exception `e_AnnouncementError` is raised if the `announcementId` is invalid.

The exception `e_ApplicationInfoError` is raised if there are unknown or invalid values for `t_ApplicationInfo`, or if the application is incompatable with the type of service being joined.

8.6.4.27 replyToInvitation()

```
void replyToInvitation (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEEAccessCommonTypes::t_InvitationId invitationId,
      in SPFEECommonTypes::t_InvitationReply reply
) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEEAccessCommonTypes::e_InvitationError,
      SPFEECommonTypes::e_InvitationReplyError
);
```

The replyToInvitation() allows the consumer to reply to an invitation, which the consumer has received. It allows the consumer to inform the retailer of his intention to join, or not, the session, or of a different location to look for the consumer. (Joining the session cannot be accomplished through this operation.)

invitationId is the identifier of the invitation.

reply is a structure which contains the information about the consumer's reply. (For details, see "Invitations and Announcements".)

The exception e_InvitationError is raised if the invitationId is invalid.

The exception e_InvitationReplyError is raised if there is an error in reply.

8.6.5 i_RetailerAnonAccess Interface

```
interface i_RetailerAnonAccess
      : i_RetailerAccess
{
      // No operations defined at present
};
```

i_RetailerAnonAccess interface allows an anonymous consumer access to services. The consumer uses it for all operations within an access session with the retailer.

This interface is returned when the consumer has established an anonymous access session with the retailer. It is returned by calling requestAnonAccess() on the i_RetailerInitial interface.

This interface inherits from i_RetailerAccess interface. The i_RetailerAccess interface is currently blank. It will contain operations which are shared between the i_RetailerNamedAccess interface, and this interface. This means that the operations offered by this interface will change in the future.

8.6.6 i_DiscoverServicesIterator Interface

```
interface i_DiscoverServicesIterator
{
      // Operations defined in the following subsections
};
```

This interface is returned by the discoverServices() operation on the i_RetailerNamedAccess interface. It is used to access remaining services, which were not returned by the discoverServices() operation.

The discoverServices() operation returns a list of services which matched some properties defined by the consumer. This interface allows the consumer to access the remaining services which were not returned by the call to discoverServices(). This is necessary because the list of services matching the properties could be very large, and include large amounts of information, potentially too much for the consumer's application to handle.

Using the discoverServices() operation, following by possibly multiple calls on the nextN() operation on this interface, allows the consumer to access all of the services matching the properties, without having to receive all of them at once.

8.6.6.1 maxLeft()

```
void maxLeft (
      out unsigned long n
) raises (
      e_UnknownDiscoverServicesMaxLeft
);
```

The maxLeft() returns the maximum number of services which will be returned through this interface. These services can be accessed through multiple calls on the nextN() operation.

maxLeft() raises the exception e_UnknownDiscoverServicesMaxLeft if it is not possible for the retailer to determine the number of maximum number of services which could be returned.

8.6.6.2 nextN()

```
void nextN (
      in unsigned long n,
      out SPFEEAccessCommonTypes::t_ServiceList services,
      out boolean moreLeft
) raises (
      SPFEECommonTypes::e_ListError
);
```

The nextN() allows the consumer to access the remaing services which were not returned by the discoverServices() operation, or by previous calls to this operation. These services can be accessed through multiple calls on the nextN() operation.

The n parameter determines the maximum number of services to be returned. The length of the services list will not exceed n.

The remaining services are returned as the t_ServiceList services. This is a sequence of t_ServiceInfo structures, which contain the t_ServiceId, t_UserServiceName (consumer's name for the service), and a sequence of service properties, t_ServiceProperties.

The moreLeft parameter is a boolean to inform the consumer if there are any remaining services, after this call to nextN().

8.6.6.3 destroy()

```
void destroy ();
```

The destroy() operation is used to inform the retailer that the consumer has finished with the i_DiscoverServicesIterator interface. It may be called at any time by the consumer, (i.e. the consumer does not has to have retrieved all the services before destroying the interface). After it has returned, the consumer will not be able to use their reference to the i_DiscoverServicesIterator interface again.

8.7 Subscription Management

This subclause is dedicated to the description of the interfaces offered by the retailer to the consumer, for supporting subscription related functionality. There exist two types of interfaces, corresponding to two types of consumers: the one accessed by the anonymous user, enabling him to become a subscriber, and the ones accessed by the subscriber, and enabling him to manage all its subscription related information. The subscription related interactions take place in a generic (service independent) fashion. Note that ancillary services can implement an OnLine Subscription facility. This being service-specific, it is considered outside the scope of this Supplement. The retailer's subscription interfaces are accessed in the context of the access session, to retrieve the list of services the consumer/subscriber/user is associated to and the corresponding service profiles, and modify the subscription data.

The main functionality offered to an anonymous user is:

– retrieval of the list of services (i.e. the ones available through this retailer);

– creation of a new subscriber.

The main functionality offered to a subscriber is:

– retrieval of the list of services, either the ones available through this retailer, or the subscribed ones;

– creation, modification, deletion and query of subscriber related information (associated end users, end user groups, etc.);

– creation, modification, deletion and query of service contracts (definition of subscribed service profiles);

– retrieval of the service profile (SAGServiceProfile) for a specific user (or terminal or NAP).

8.7.1 Subscription Management Type Definitions

In this subclause, the IDL definition of the information required to handle subscriptions, subscribers and end-users in a provider domain is included. This will allow to understand more clearly the interface descriptions.

Figure 8-4 represents mainly the relationship between a service and a subscriber, described in terms of a number of service profiles (service template, subscription profile and SAG service profile).

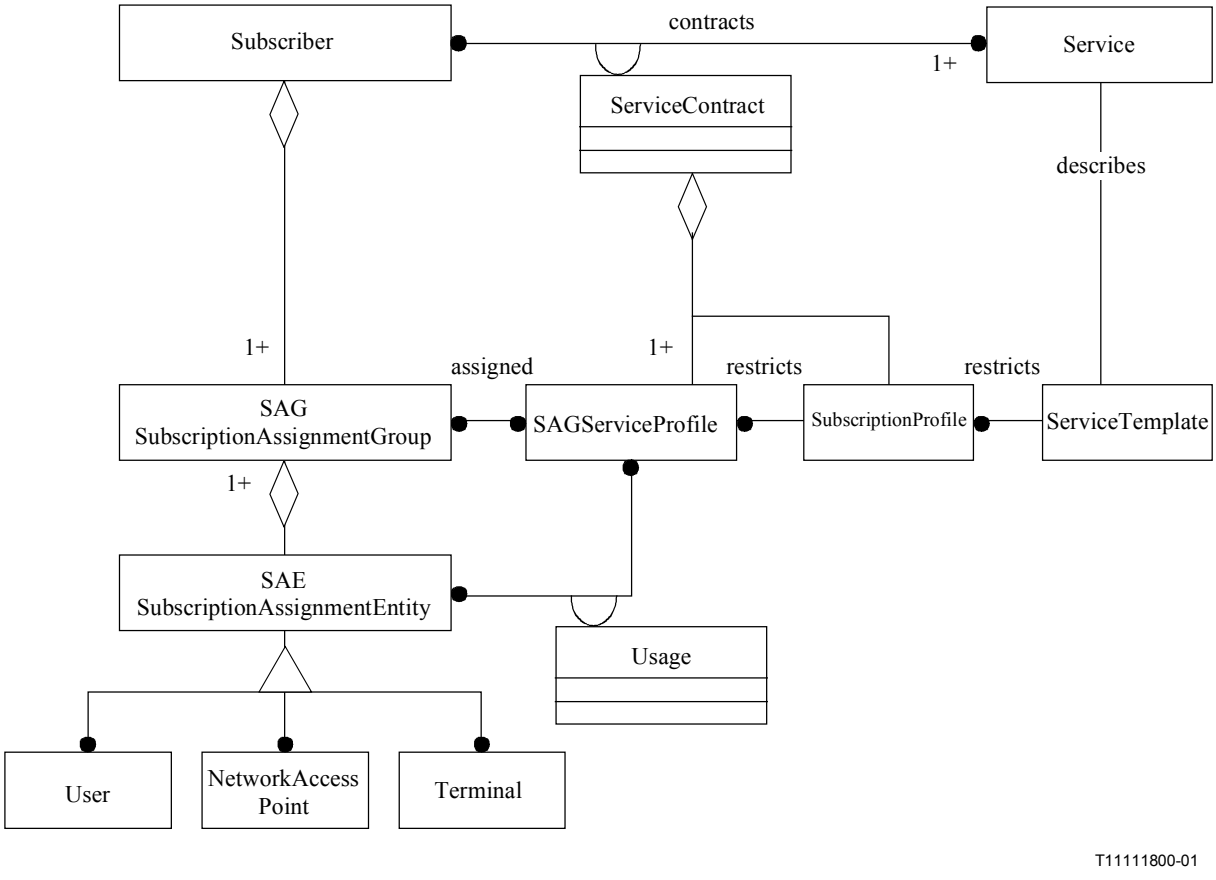


Figure 8-4 – Subscription management information model

A Subscriber contracts a number of Services (at least one to be considered as such). The information associated with a subscriber is:

```
struct t_Subscriber {
   t_AccountNumber                  accountNumber;
   SPFEECommonTypes::t_UserId       subscriberName;
   t_Person                         identificationInfo;
   t_Person                         billingContactPoint;
   string                           RatePlan;
   any                              paymentRecord;
   any                              credit;
};
```

The `accountNumber` is generated by the retailer and is unique in its domain. It is used inside the retailer domain to identify the subscriber and, probably, in the bills as well. The `subscriberName` is the name the subscriber wants to be named by[41]. It will be usually more user-friendly than the `accountNumber`. There will be a one-to-one mapping between these two identifiers. The field `identificationInfo` stores information like subscriber name, address, etc. The `billingContactPoint` keeps the information required to send the invoices for billing. The `paymentRecord` contains information about last paid bills to check the billing status. The `credit` field stores information about deposits, credits granted to the subscriber, etc.

The agreed Service Contract defines the conditions of the service provision for each of the service subscriptions. It is defined as:

```
struct t_ServiceContract {
   SPFEEAccessCommonTypes::t_ServiceId      serviceId;
   t_AccountNumber                          accountNumber;
   short                                    maxNumOfServiceProfiles;
   t_DateTime                               actualStart;
   t_DateTime                               requestedStart;
   t_Person                                 requester;
   t_Person                                 technicalContactPoint;
   t_AuthLimit                              authorityLimit;
   t_SubscriptionProfile                    subscriptionProfile;
   t_SagServiceProfileList                  sagServiceProfileList;
};
```

The `serviceId` and the `accountNumber`, together, identify uniquely a service contract. The profiles are the main part of the service contract. Other fields provide additional information about the contract (starting date, requested starting date, requester, technical contact point, etc.).

A Service Template describes the characteristics of the Service accessible through the retailer.

```
struct t_ServiceTemplate {
   SPFEEAccessCommonTypes::t_ServiceId          serviceInstanceId;
   SPFEEAccessCommonTypes::t_UserServiceName    serviceInstanceName;
   t_ServiceIdList                              requiredServices;
   t_ServiceDescription                         serviceDescription;
};
```

The Service Description contains the characteristics of a generic service type. It is reused in the service template to describe the characteristics of a specific service instance (particular implementation of a service type) and in the service profiles to represent the characteristics of the service contracted by a subscriber (for the whole set of associated users or for a group of them).

[41] It might be used to generate user identifiers. For instance, user X subscribed with retailer A could be given an identifier like X@A.

```
struct t_ServiceDescription {
   SPFEEAccessCommonTypes::t_ServiceId           serviceTypeId;
   SPFEEAccessCommonTypes::t_UserServiceName     serviceTypeName;
   t_ParameterList                               serviceCommonParams;
   t_ParameterList                               serviceSpecificParams;
};
```

The Parameter List consists of a sequence of triples composed of parameter name, parameter configurability and parameter value.

```
typedef string  t_ParameterName;
enum t_ParameterConfigurability { FIXED_BY_PROVIDER,
   CONFIGURABLE_BY_SUBSCRIBER, CUSTOMIZABLE_BY_USER };

typedef any     t_ParameterValue;

struct t_Parameter {
   t_ParameterName                    name;
   t_ParameterConfigurability         configurability;
   t_ParameterValue                   value;
};

typedef sequence<t_Parameter> t_ParameterList;
```

The retailer may give the subscriber the option to select specific service parameters to apply to all its associated entities[42] – Subscription Profile – or to a group of them – SAG Service Profile –, reducing the alternatives (restricts association in Figure 8-4) given in the service template. These profiles are the main part of the service contract.

```
typedef string t_ServiceProfileId;
struct t_ServiceProfile {
   t_ServiceProfileId             spId;
   t_ServiceDescription           serviceDescription;
};
typedef t_ServiceProfile t_SagServiceProfile;
typedef t_ServiceProfile t_SubscriptionProfile;
```

A set of entities, Users, Terminals or NAPs, can be associated to a subscriber. Let's call them Subscription Assignment Entities (SAEs).

```
enum t_entityType {user, terminal, nap};

/* Entity Id identifies uniquely a SAE inside the provider domain. */
union t_entityId switch (t_entityType) {
   case user:       SPFEECommonTypes::t_UserId             userId;
   case terminal:   SPFEEAccessCommonTypes::t_TerminalId   terminalId;
   case nap:        SPFEEAccessCommonTypes::t_NAPId        napId;
};

typedef sequence<t_entityId> t_entityIdList;

/* A SAE is characterized by an identifier, a name and a set of properties. */
struct t_Sae {
   t_entityId                              entityId;
   string                                  entityName;
   SPFEECommonTypes::t_PropertyList        properties;
};
```

The subscriber may not want to grant all of them with the same service characteristics (or privileges). For this reason, the subscriber can group them in a set of Subscription Assignment Groups (SAG):

---

[42] Users, terminals or network access points.

```
typedef short    t_SagId;

/* A SAG is characterized by its identifier, a textual description of the
* group and the list of entities composing it. */
struct t_Sag {
    t_SagId                    sagId;
    string                     sagDescription;
    t_entityIdList             entityList;
};
```

The subscriber can then assign particular service profiles (SAG Service Profile) to each group. The main reason for using this grouping is to ease the subscription process (assignment of profiles to users) in subscriber domains where end-users are naturally classified in categories, organizational or geographical areas, etc., requiring the same service usage privileges. The only restriction to apply is that every SAE must be assigned to one and only one SAG Service Profile for every service.

For every subscriber a default SAG is created with SAGId (0). Every SAE is always assigned to this SAG even if it is assigned to other particular SAG. If a user is removed from any SAG, it will still be associated to this SAG by default. The SAG by default can not be associated to service profiles and users can not be assigned to this SAG explicitly (they are implicitly assigned to it on creation).

It is also possible to assign and remove individual SAEs to/from service profiles. This is specially interesting in small subscriber organizations (like residential customers), where the definition of user groups is not strictly needed and does not help the subscriber in the subscription management. Additionally, this provides a lot of flexibility in the service profile assignment, as some users in a group can be discriminated for the access to a specific service, without the need of removing them from the group or defining a new group.

A structure like t_UsagePermit may help in the definition of these restrictions:

```
enum t_UsagePermitFlag {USAGE_ALLOWED, USAGE_DISALLOWED};
struct t_UsagePermit {
    t_entityId                 entityId;
    t_ServiceProfileId         serviceProfileId;
    t_UsagePermitFlag          flag;
};
```

The following sections describe the interfaces offered by the retailer to the consumer for supporting the subscription functionality. These are a set of service specific interfaces offered through Ret reference point for the online management of service subscriptions. There are different interfaces for every user type (subscribers or retailer operators).

8.7.2 i_SubscriberSubscriptionMgmt

```
// module SPFEERetSubscriberSubscriptionMgmt
interface i_SubscriberSubscriptionMgmt
{
};
```

This interface is dedicated to subscribers. It provides operations for subscribing with the retailer, contracting services and defining subscriber and service contract information. It allows the subscription and service contract cancellation and modification and the query of all the subscriber related information.

Operations for applying for service contracts, subscriptions and cancellations:

- **listServices**() – It returns the list of services provided by the retailer.
- **subscribe**() – It allows to create a subscription contract with the retailer. As input parameters it has the subscriber information and a list of services the subscriber is willing to subscribe to. It returns a subscriber identifier and a list of service contract identifiers. These will be used in the following for making reference to specific service contracts.
- **unsubscribe**() – It allows to delete a (list of) service contract(s) or the whole relationship with the retailer.
- **contractService**() – It subscribes a subscriber to a service and returns an interface reference where he can define the service contract (`i_ServiceContractInfoMgmt`).
- **listSubscribedServices**() – It returns the list of contracted services (just the identifiers) and related service contract identifiers. If a user is specified, it returns the list of services granted to that specific user by the subscriber. The subscriber identifier (account number) is indicated as an input parameter.

Operations for handling subscriber information:

- **listSAEs**() – It returns the list of entities associated to the subscriber. If a (list of) SAG identifier(s) is specified, it returns only the users assigned to that(those) SAG(s).
- **listSAGs**() – It returns the list of SAGs (ids) for that subscriber.
- **getSubscriberInfo**() – It returns the information about the subscriber.
- **createSAEs**() – it creates the entities specified as a parameter returning an identifier for each of them.
- **deleteSAEs**() – it deletes the entities specified as a parameter. It removes any existing assignment to SAGs these entities could have.
- **createSAGs**() – it creates a (number of) SAG(s). A list of entities for every SAG can be specified. A SAG identifier is returned to ease further management.
- **assignSAEs**() – It assigns a list of entities to a SAG.
- **removeSAEs**() – It removes a list of entities from a SAG.
- **setSubscriberInfo**() – It modifies the information about the subscriber.

Operations for defining, modifying and querying service contracts:

- **getServiceTemplate**() – It returns the template for service profile definition for the specified service.
- **defineServiceContract**() – It allows to define the service contract for a specific service. This contract includes, amongst other contractual information, the set of service profiles composing the service contract, namely the subscription profile (applicable to all users) and the set of SAG service profiles (each one applicable to a SAG and consistent with the subscription profile). It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service contracts.
- **defineServiceProfiles**() – It allows to define a set of service profiles for a service contract, namely the subscription profile and the set of SAG service profiles. It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service profiles.
- **deleteServiceProfiles**() – It deletes a service profile.
- **getServiceContractInfo**() – It returns the information related to the service contract which identifier is passed as parameter. If a list of SAG profile identifiers is specified, the set of associated SAG service profiles is returned.

Operations for authorization and activation of service profiles:

- **`assignServiceProfile`**() – It associates a list of SAEs and SAGs with a SAGServiceProfile. If the service profile is active, the SAEs (the explicitly stated and the ones included in the SAGs) will be able to use the service. These SAEs' access components will be notified and the SAG service profile will be make available for them. From this profile, the SAE will be able to customize its own user service profile using the service profile management service.
- **`removeServiceProfile`**() – It disassociates a list of SAEs and SAGs from a SAGServiceProfile. The specified SAEs (individually specified or inside a SAG) will no longer be able to use the service, unless associated with another active service profile.
- **`activateServiceProfiles`**() – It activates a list of SAG service profiles making them available for use. Only SAEs and SAGs assigned to an active Service Profile can make use of the service.
- **`deactivateServiceProfiles`**() – It deactivates a list of SAG service profiles. Users (or SAEs) assigned to these service profiles will not be able to use the service.

8.7.3 i_RetailerSubscriptionMgmt

```
// module SPFEERetRetailerSubscriptionMgmt
interface i_RetailerSubscriptionMgmt
{
};
```

This interface provides full capabilities for subscribing customers, add, modify, cancel and query service contracts and adding, removing, modifying and querying subscriber information. In general, it provides access to the whole subscription database. It is thus a superset of the previous interface, dedicated to a consumer of the type retailer operator.

Additional operations are:

- **`listSubscribers`**() – It returns the list of subscribers (identifiers). If a service Id is specified, it returns the list of subscribers for that service.
- **`listServiceContracts`**() – It returns the list of service contracts (identifiers). If a service Id is specified, it returns the list of service contracts for that service. If a subscriber is specified, it acts as the `listSubscribedServices` in the `i_SubscriberInfoMgmt` interface for that particular subscriber.
- **`listUsers`**() – It returns the list of users (identifiers) for a specified service.

9 Complete IDL specifications

9.1 Common Definitions IDLs

9.1.1 SPFEECommonTypes.idl

```
// File SPFEECommonTypes.idl
#ifndef spfeecommontypes_idl
#define spfeecommontypes_idl

module SPFEECommonTypes {

// ElementIds (mainly used in usage part)
```

```
// Element types
enum t_ElTypes {
   SpfeeParty,            // see also t_PartyId
   SpfeeResource, SpfeeMember,
   SpfeeGroup, SpfeeMemberGroup, SpfeePartyGroup, SpfeeResourceGroup,
   SpfeeRelation, SpfeeRelationGroup, SpfeeControlRelation,
   SpfeeStreamBinding, SpfeeStreamFlow, SpfeeStreamInterface, SpfeeSFEP
};

// Element Identifier (elements in a service session)
typedef unsigned long t_ElId;

// Element Type Identifiers
typedef t_ElTypes t_ElTypeId;

// Overall element Identifier
struct t_ElementId
{
   t_ElId id;
   t_ElTypeId elType;
};

typedef sequence <t_ElementId> t_ElementIdList;

// t_PropertyList
// list of properties, (name value pairs).
// Used in many operations to allow a list of as yet undefined
// properties, and values, to be sent.
//

typedef string t_PropertyName;
typedef sequence<t_PropertyName> t_PropertyNameList;
typedef any t_PropertyValue;

struct t_Property {
   t_PropertyName name;
   t_PropertyValue value;
};

typedef sequence<t_Property> t_PropertyList;

enum t_HowManyProps {none, some, all};

union t_SpecifiedProps switch (t_HowManyProps) {
   case some: t_PropertyNameList prop_names;
   case none: octet dummy1;
   case all: octet dummy2;
};

typedef string Istring;

// enum t_ReferenceSort {
// ObjectRef,
// StringifiedReference
// };

// union t_Reference switch(t_ReferenceSort) {
// case ObjectRef: Object IRef;
// case StringifiedReference: SPFEECommonTypes::Istring IORef;
// };

enum t_WhichProperties {
   NoProperties,         // don't can ignore all the properties
   SomeProperties,       // match at least one property (name & value)
   SomePropertiesNamesOnly, // check name only (ignore value)
```

```
   AllProperties,        // match all properties (name & value)
   AllPropertiesNamesOnly    // check name only (ignore value)
};

struct t_MatchProperties {
   t_WhichProperties whichProperties;
   t_PropertyList properties;
};

typedef /*CORBA::*/Object t_Interface;

typedef string t_InterfaceTypeName;
typedef sequence<t_InterfaceTypeName> t_InterfaceTypeList;
typedef t_PropertyList t_InterfaceProperties;

struct t_InterfaceStruct {
        t_InterfaceTypeName itfType;
        Object ref;
          // if NULL: use getInterface(type)
          // to get the reference
        t_InterfaceProperties properties;
          // interface type specific properties
          // interpreted by the session.
};

typedef sequence<t_InterfaceStruct> t_InterfaceList;

typedef unsigned long t_InterfaceIndex;
typedef sequence<t_InterfaceIndex> t_InterfaceIndexList;

// when registering multiple interfaces need to match index vs itfType &
props:
struct t_RegisterInterfaceStruct {
      t_InterfaceTypeName itfType;// set before call to registerInterfaces
      Object ref;
t_InterfaceProperties properties;  // as above
t_InterfaceIndex index;           // set on return
};

typedef sequence<t_RegisterInterfaceStruct> t_RegisterInterfaceList;

// Session Models

// t_SessionModelName name:
// defined names
//    "SPFEEServiceSessionModel"    SPFEE Service Session Model
//    "SPFEECommSessionModel"       SPFEE Communication Session Model
// No other names defined at present
//
//  (previous  versions  of  Ret-RP  used  "SPFEESessionModel"  and  "SPFEE
//    Session Model" for the SPFEE Service Session Model (previously named
//    SPFEE Session Model)

typedef string t_SessionModelName;

typedef sequence<t_SessionModelName> t_SessionModelNameList;

// t_SessionModelProperties properties:
//
// t_SessionModelName name: "SPFEEServiceSessionModel"
// defined Property names:
// name:       "FEATURE SETS"
// value:      t_FeatureSetList
// No other names defined at present
```

```
// t_SessionModelName name: "SPFEECommSessionModel"
// defined Property names:
// name:       "FEATURE SETS"
// value:      t_FeatureSetList
//
// No feature sets are defined for the TIACommSessionModel at present.
//
// No other names defined at present

typedef t_PropertyList t_SessionModelProperties;

struct t_SessionModel {
   t_SessionModelName name;                // reserved names defined below
   t_SessionModelProperties properties;    // properties defined above
};

typedef sequence<t_SessionModel> t_SessionModelList;

enum t_WhichSessionModels {
   NoSessionModels,    // can ignore all the SessionModels
   SomeSessionModels,  // match at least one SessionModel (name & value)
   SomeSessionModelsNamesOnly,     // check name only (ignore value)
   AllSessionModels,   // match all SessionModels (name & value)
   AllSessionModelsNamesOnly   // check name only (ignore value)
};

struct t_SessionModelReq {
   t_WhichSessionModels which;
   t_SessionModelList sessionModels;
};

typedef string t_FeatureSetName;

struct t_FeatureSet {
   t_FeatureSetName name;
   t_InterfaceList itfs;
                // can return ref or NULL
};

typedef sequence<t_FeatureSet> t_FeatureSetList;

// User Info

typedef Istring t_UserId;
typedef Istring t_UserName;
typedef t_Property t_UserProperty;
typedef t_PropertyList t_UserProperties;
// Property Names defined for t_UserProperties:
// name: "PASSWORD"
// value: string
// use:  user password, as a string.

// name:  "SecurityContext"
// value: opaque
// use:   to carry a retailer specific security context
//        e.g. could be used for an encoded user password.

struct t_UserDetails {
   t_UserId id;
   t_UserProperties properties;
};

typedef Istring t_UserCtxtName;
typedef sequence<t_UserCtxtName> t_UserCtxtNameList;
```

```
typedef sequence<octet, 16> t_ParticipantSecretId;
typedef t_ElId t_PartyId;    // corresponds to SpfeeParty enum t_ElTypes
typedef sequence<t_PartyId> t_PartyIdList;

// SessionId
// A SessionId is generated by a UA when a new session is started.
// This Id is unique within a UA, and can be used to identify a
// session to the UA.
// User's joining a session will have a different SessionId generated
// by their UA for the session.

typedef unsigned long t_SessionId;
typedef sequence<t_SessionId> t_SessionIdList;
typedef t_PropertyList t_SessionProperties;

// Invitation and Announcements
//
enum t_InvitationReplyCodes { // Based on MMUSIC replys
   SUCCESS,            // user agrees to participate
   UNSUCCESSFUL,             // couldn't contact user
   DECLINE,            // user declines
   UNKNOWN,            // user is unknown
   ERROR,              // for some unknown reason
   FORBIDDEN,                // authorisation failure
   RINGING,            // user is being contacted
   TRYING,                   // some further action is being taken
   STORED,                   // invitation is stored
   REDIRECT,           // try this different address
   NEGOTIATE,                // alternatives described in properties
      // Not MMUSIC replys, can be treated as UNSUCCESSFUL
   BUSY,               // couldn't contact because busy
   TIMEOUT                   // timed out while trying to contact
};

typedef t_PropertyList t_InvitationReplyProperties;

struct t_InvitationReply {
   t_InvitationReplyCodes reply;
   t_InvitationReplyProperties properties;
};

typedef t_PropertyList t_AnnouncementProperties;

struct t_SessionAnnouncement {
   t_AnnouncementProperties properties;
};

typedef sequence<t_SessionAnnouncement> t_AnnouncementList;

// Exceptions

enum t_PropertyErrorCode {
   UnknownPropertyError,
   InvalidProperty,
   // UnknownPropertyName: If the server receives a property name
   // it doesnot know, it can raise an exception, using this code.
   // However, servers may decide to ignore a property with an
   // unknown property name, and not raise an exception.
   UnknownPropertyName,
   InvalidPropertyName,
   InvalidPropertyValue,
   NoPropertyError           // the Property is not in error
};
```

```
// defined so it can be used in other exceptions
struct t_PropertyErrorStruct {
   t_PropertyErrorCode errorCode;
   t_PropertyName name;
   t_PropertyValue value;
};

exception e_PropertyError {
   t_PropertyErrorCode errorCode;
   t_PropertyName name;
   t_PropertyValue value;
};

enum t_InterfacesErrorCode {
   UnknownInterfacesError,
   InvalidInterfaceTypeName, // Thats not a valid i/f type name
   InvalidInterfaceRef,
   InvalidInterfaceProperty,
   InvalidInterfaceIndex
};

// must remain consistent with e_InterfacesError
struct t_InterfacesErrorStruct {
   t_InterfacesErrorCode errorCode;
   t_InterfaceTypeName itfType;
   t_PropertyErrorStruct propertyError;
      //PropertyError, if errorCode= InvalidInterfaceProperty
};

exception e_InterfacesError {
   t_InterfacesErrorCode errorCode;
   t_InterfaceTypeName itfType;
   t_PropertyErrorStruct propertyError;
      //PropertyError, if errorCode= InvalidInterfaceProperty
};

enum t_RegisterErrorCode {
   UnableToRegisterInterfaceType
};

exception e_RegisterError {
   t_RegisterErrorCode errorCode;
   t_InterfaceTypeName itfType;
   t_InterfaceProperties properties;
};

enum t_UnregisterErrorCode {
   UnableToUnregisterInterface
};

exception e_UnregisterError {
   t_UnregisterErrorCode errorCode;
   t_InterfaceIndexList indexes;   // can unregister multiple itfs
};

enum t_SessionModelErrorCode {
   UnknownSessionModelError,
   InvalidSessionModelName,     // Thats not a valid i/f type name
   SessionModelNotSupported,
   InvalidFeatureSetName,
   FeatureSetNotSupported,
   InvalidFeatureSetInterfaceType
};
```

```
exception e_SessionModelError {
   t_SessionModelErrorCode errorCode;
   t_SessionModelName sessionModelName;
   t_FeatureSetName featureSetName;// Only for FeatureSet errs
   t_InterfaceTypeName itfType;    // Only for FeatureSet errs
};

enum t_UserDetailsErrorCode {
   InvalidUserName,
   InvalidUserProperty
};

exception e_UserDetailsError {
   t_UserDetailsErrorCode errorCode;
   t_UserName name;
   t_PropertyErrorStruct propertyError;
           // Return the properties in error
};

enum t_ListErrorCode {
   ListUnavailable
};

exception e_ListError {
   t_ListErrorCode errorCode;
};

enum t_InvitationReplyErrorCode {
   InvalidInvitationReplyCode,
   InvitationReplyPropertyError
};

exception e_InvitationReplyError {
   t_InvitationReplyErrorCode errorCode;
   t_PropertyErrorStruct propertyError;
};

}; // end module SPFEECommonTypes

#endif
```

9.1.2 SPFEEAccessCommonTypes.idl

```
// File SPFEEAccessCommonTypes.idl

#ifndef spfeeaccesscommontypes_idl
#define spfeeaccesscommontypes_idl

#include "SPFEECommonTypes.idl"

module SPFEEAccessCommonTypes {

// User Info

// The following Login-Password combination may be used
// for non-CORBA Security-compliant systems, which still
// relies on a traditional, login name & password combination.
// It is mainly provided for compability reasons for the legacy
// systems, and is not expected to be used with the CORBA compliant
// part at the same time.
// legacy authentication

typedef SPFEECommonTypes::Istring t_UserPassword;
```

```
struct t_UserInfo {
   SPFEECommonTypes::t_UserId userId;
   SPFEECommonTypes::t_UserName name;
   SPFEECommonTypes::t_UserProperties userProperties;
};

// Access Session

typedef sequence <octet, 16> t_AccessSessionSecretId;
   // 128b array generated by Retailer(should be self checking)
   // (is big enough to hold the GUID favored by DCE & DCOM)
   // (Globally Unique IDentifier)

typedef unsigned long t_AccessSessionId;

enum t_WhichAccessSession {
   CurrentAccessSession,
   SpecifiedAccessSessions,
   AllAccessSessions
};

typedef sequence<t_AccessSessionId> t_AccessSessionIdList;

// Implementation Note:
// Orbix does not allow the creator of a union to set the
// discriminator (switch tag). If true, this union requires
// dummy cases for the other enums of t_WhichAccessSession.

union t_SpecifiedAccessSession switch (t_WhichAccessSession) {
   case SpecifiedAccessSessions: t_AccessSessionIdList asIdList;
   case CurrentAccessSession: octet dummy1;
   case AllAccessSessions: octet dummy2;
           // dummy var's values should not be processed
};

typedef SPFEECommonTypes::t_PropertyList t_AccessSessionProperties;

struct t_AccessSessionInfo {
   t_AccessSessionId id;
   SPFEECommonTypes::t_UserCtxtName ctxtName;
   t_AccessSessionProperties properties;
};

typedef sequence<t_AccessSessionInfo> t_AccessSessionList;

// Terminal Info

typedef string t_TerminalId;
typedef sequence<string> t_NAPId;


typedef sequence<t_NAPId> t_NAPIdList;

typedef string t_NAPType;

typedef SPFEECommonTypes::t_PropertyList t_TerminalProperties;

// t_TerminalProperties properties:
//
// defined Property names:
// name:   "TERMINAL INFO"
// value:  t_TerminalInfo
```

```
// name:    "APPLICATION INFO LIST"
// value:   t_ApplicationInfoList
// Applications on the terminal

// No other names defined at present

// t_TermType
// DESCRIPTION:
// List of terminal types.
// COMMENTS:
//  - This list can be expanded.

enum t_TerminalType {
   PersonalComputer, WorkStation, TVset,
   Videotelephone, Cellularphone, PBX, VideoServer,
   VideoBridge, Telephone, G4Fax
};

// t_TermInfo
// DESCRIPTION:
// This structure contains information related to a specific terminal
// COMMENTS:
// To be defined further.

struct t_TerminalInfo {
   t_TerminalType terminalType;
   string operatingSystem; // includes the version
   SPFEECommonTypes::t_PropertyList networkCards;
   SPFEECommonTypes::t_PropertyList devices;
   unsigned short maxConnections;
   unsigned short memorySize;
   unsigned short diskCapacity;
};

// Provider Agent Context

struct t_TerminalConfig {
   t_TerminalId terminalId;
   t_TerminalType terminalType;
   t_NAPId napId;
   t_NAPType napType;
   t_TerminalProperties properties;
};

// Service Types
//

typedef unsigned long t_ServiceId;

typedef sequence<t_ServiceId> t_ServiceIdList;

typedef SPFEECommonTypes::Istring t_UserServiceName;

typedef SPFEECommonTypes::t_PropertyList t_ServiceProperties;

struct t_ServiceInfo {
   t_ServiceId id;
   t_UserServiceName name;
   t_ServiceProperties properties;
};

typedef sequence<t_ServiceInfo> t_ServiceList;
```

```
// Session State
// State of the session as seen from the users point of view

enum t_UserSessionState {
   UserUnknownSessionState,    // Session State is not known
   UserActiveSession,
   UserSuspendedSession,     // Session has been suspended
   UserSuspendedParticipation,     // User has suspendedParticipation
          // but is continuing in his absence.
          // (may have been quit subsequently)
   UserInvited,        // User has been invited to join
   UserNotParticipating            // User is not in the session
};

// Session Info

typedef SPFEECommonTypes::Istring t_SessionPurpose;

struct t_SessionOrigin {
   SPFEECommonTypes::t_UserId userId; // user creating the session
   SPFEECommonTypes::t_SessionId sessionId;
           // id (unique to originating user)
};

struct t_SessionInfo {
   SPFEECommonTypes::t_SessionId id;     // my session id,
                // unique to UA. (scope by UA).
   t_SessionPurpose purpose;

   SPFEECommonTypes::t_ParticipantSecretId secretId;
   SPFEECommonTypes::t_PartyId myPartyId;
   t_UserSessionState state;

   SPFEECommonTypes::t_InterfaceList itfs;
   SPFEECommonTypes::t_SessionModelList sessionModels;
   SPFEECommonTypes::t_SessionProperties properties;
};

// for listing active/suspended sessions
typedef sequence<t_SessionInfo> t_SessionList;

// Invitations and Announcements.

typedef unsigned long t_InvitationId;
typedef SPFEECommonTypes::Istring t_InvitationReason;

struct t_InvitationOrigin {
   SPFEECommonTypes::t_UserId userId;    // user creating the invitation
   SPFEECommonTypes::t_SessionId sessionId;
      // so they which session they invited you from, if you contact them
};

struct t_SessionInvitation {
   t_InvitationId id;

   SPFEECommonTypes::t_UserId inviteeId;   // id of invited user, so you
                                           can check
                // the invitation was for you.
t_SessionPurpose purpose;
t_ServiceInfo serviceInfo;
t_InvitationReason reason;
t_InvitationOrigin origin;
SPFEECommonTypes::t_PropertyList invProperties;
};
```

```
typedef sequence<t_SessionInvitation> t_InvitationList;

typedef unsigned long t_AnnouncementId;

// Start Service Properties and  Application Info.

typedef SPFEECommonTypes::Istring t_AppName;
typedef SPFEECommonTypes::Istring t_AppVersion;
typedef SPFEECommonTypes::Istring t_AppSerialNum;
typedef SPFEECommonTypes::Istring t_AppLicenceNum;

struct t_ApplicationInfo {
   t_AppName name;
   t_AppVersion version;
   t_AppSerialNum serialNum;
   t_AppLicenceNum licenceNum;
   SPFEECommonTypes::t_PropertyList properties;
   SPFEECommonTypes::t_InterfaceList itfs;
   SPFEECommonTypes::t_SessionModelList sessionModels;
};

// t_StartServiceUAProperties properties:
// properties to be interpreted by the User Agent, when starting a service
//
// defined Property names:
// None defined at present
typedef SPFEECommonTypes::t_PropertyList t_StartServiceUAProperties;

// t_StartServiceSSProperties properties:
// properties to be interpreted by the Service Session, when starting a
service
//
// defined Property names:
// None defined at present
typedef SPFEECommonTypes::t_PropertyList t_StartServiceSSProperties;


// Exceptions

enum t_AccessErrorCode {
        UnknownAccessError,
        InvalidAccessSessionSecretId,
        AccessDenied,
        SecurityContextNotSatisfied
};

exception e_AccessError {
        t_AccessErrorCode errorCode;
};

enum t_UserPropertiesErrorCode {
   InvalidUserPropertyName,
   InvalidUserPropertyValue
};

exception e_UserPropertiesError {
   t_UserPropertiesErrorCode errorCode;
   SPFEECommonTypes::t_UserProperty userProperty;
};
```

```
enum t_SpecifiedAccessSessionErrorCode {
   UnknownSpecifiedAccessSessionError,
   InvalidWhichAccessSession,
   InvalidAccessSessionId
};

exception e_SpecifiedAccessSessionError {
   t_SpecifiedAccessSessionErrorCode errorCode;
   t_AccessSessionId id;           // Invalid AccessSessionId
};

enum t_InvitationErrorCode {
   InvalidInvitationId
};

exception e_InvitationError {
   t_InvitationErrorCode errorCode;
};
};
#endif
```

9.2 User and Provider General IDLs

9.2.1 SPFEEUserInitial.idl

```
// File SPFEEUserInitial.idl
#ifndef spfeeuserinitial_idl
#define spfeeuserinitial_idl

#include "SPFEECommonTypes.idl"
#include "SPFEEAccessCommonTypes.idl"

module SPFEEUserInitial {

// requestAccess() types

typedef string t_ProviderId;

// inviteUserWithoutAccessSession() types


enum t_AccessReplyCodes {
   SUCCESS,      // user agrees to initiate an access session
   DECLINE,      // user declines to initiate an access session
   FAILED,       // for some unknown reason
   FORBIDDEN     // authorisation failure
};

typedef SPFEECommonTypes::t_PropertyList t_AccessReplyProperties;

struct t_AccessReply {
   t_AccessReplyCodes reply;
   t_AccessReplyProperties properties;
};

interface i_UserInitial
{
// behaviour
// behaviourText
// "This interface is provided to allow a Provider to invite
// a user to a session outside of an access session; and request
// the establishment of an access session";
// usage
// " ";
```

```
    void requestAccess (
        in t_ProviderId providerId,
        out t_AccessReply reply
    );

    void inviteUserOutsideAccessSession (
        in t_ProviderId providerId,
        in SPFEEAccessCommonTypes::t_SessionInvitation invitation,
        out SPFEECommonTypes::t_InvitationReply reply
    );

    void cancelInviteUserOutsideAccessSession (
        in t_ProviderId providerId,
        in SPFEEAccessCommonTypes::t_InvitationId id
    ) raises (
        SPFEEAccessCommonTypes::e_InvitationError
    );

};   // i_UserInitial
};
#endif
```

9.2.2 SPFEEUserAccess.idl

```
// File SPFEEUserAccess.idl

#ifndef spfeeuseraccess_idl
#define spfeeuseraccess_idl

#include "SPFEECommonTypes.idl"
#include "SPFEEAccessCommonTypes.idl"

module SPFEEUserAccess {

typedef SPFEECommonTypes::t_PropertyList t_CancelAccessSessionProperties;

interface i_UserAccessGetInterfaces {

// behaviour
// behaviourText
// "This interface allows the provider  domain to get
// interfaces exported by this user domain."

// usage
// "This interface is not to be exported across
// Ret RP. It is inherited into the exported interfaces."

    void getInterfaceTypes (
        out SPFEECommonTypes::t_InterfaceTypeList itfTypes
    ) raises (
        SPFEECommonTypes::e_ListError
    );

    void getInterface (
        in SPFEECommonTypes::t_InterfaceTypeName itfType,
        in SPFEECommonTypes::t_MatchProperties desiredProperties,
        out SPFEECommonTypes::t_InterfaceStruct itf
    ) raises (
        SPFEECommonTypes::e_InterfacesError,
        SPFEECommonTypes::e_PropertyError
    );
```

```
   void getInterfaces (
      out SPFEECommonTypes::t_InterfaceList itfs
   ) raises (
      SPFEECommonTypes::e_ListError
   );

}; // i_UserAccessGetInterfaces

interface i_UserAccess
   : i_UserAccessGetInterfaces
{
// behaviour
// behaviourText
// "This interface is provided to a UA to perform actions during an
// access session. It inherits from i_UserAccessGetInterfaces to
// allow the retailer to ask for other interfaces exported by the
// consumer domain. (The retailer cannot register his own // // /
// interfaces!)";
   // usage
   // " ";


// DRAFT: this operation is draft only, any feedback on this operation is
//        most welcome.
       void cancelAccessSession(
          in t_CancelAccessSessionProperties options
   );

}; // i_UserAccess

interface i_UserInvite
{
// behaviour
// behaviourText
// "This interface is provided to a UA, to invite the user to:
//     - join a service session
//     - request an access session
// ";
// usage
//  " ";

// inviteUser() types

   void inviteUser (
      in  SPFEEAccessCommonTypes::t_SessionInvitation invitation,
      out SPFEECommonTypes::t_InvitationReply reply
      )
   raises (SPFEEAccessCommonTypes::e_InvitationError)

   ;

   void cancelInviteUser (
      in SPFEECommonTypes::t_UserId inviteeId,
      in SPFEEAccessCommonTypes::t_InvitationId id
   ) raises (
      SPFEEAccessCommonTypes::e_InvitationError
   );

}; // i_UserInvite


interface i_UserTerminal {
```

```
// behaviour
// behaviourText
// "This interface is provided to a UA, to gain information about the
// terminal context.";
   // usage
   // " ";

// getTerminalInfo() types


   void getTerminalInfo(
      out SPFEEAccessCommonTypes::t_TerminalInfo terminalInfo
   );
}; // i_UserTerminal

interface i_UserAccessSessionInfo
{
// ...AccessSessionInfo() types

oneway void newAccessSessionInfo (
       in SPFEEAccessCommonTypes::t_AccessSessionInfo accessSession
   );

oneway void endAccessSessionInfo (
       in SPFEEAccessCommonTypes::t_AccessSessionId asId
   );

oneway void cancelAccessSessionInfo (
       in SPFEEAccessCommonTypes::t_AccessSessionId asId
   );

oneway void newSubscribedServicesInfo (
       in SPFEEAccessCommonTypes::t_ServiceList services
   );

}; // i_UserAccessSessionInfo

interface i_UserSessionInfo
{
oneway void newSessionInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );

oneway void endSessionInfo (
       in SPFEECommonTypes::t_SessionId sessionId
   );

oneway void endMyParticipationInfo (
       in SPFEECommonTypes::t_SessionId sessionId
   );

oneway void suspendSessionInfo (
       in SPFEECommonTypes::t_SessionId sessionId
   );

oneway void suspendMyParticipationInfo (
       in SPFEECommonTypes::t_SessionId sessionId
   );

oneway void resumeSessionInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );
```

```
oneway void resumeMyParticipationInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );

oneway void joinSessionInfo (
       in SPFEEAccessCommonTypes::t_SessionInfo session
   );

}; // i_UserSessionInfo

};

#endif
```

9.2.3 SPFEEProviderInitial.idl

```
// File SPFEEProviderInitial.idl
#ifndef spfeeproviderinitial_idl
#define spfeeproviderinitial_idl

#include "SPFEECommonTypes.idl"
#include "SPFEEAccessCommonTypes.idl"

module SPFEEProviderInitial {

enum t_AuthenticationStatus {
   SecAuthSuccess,
   SecAuthFailure,
   SecAuthContinue,
   SecAuthExpired
};

typedef unsigned long t_AuthMethod;


typedef SPFEECommonTypes::t_PropertyList t_AuthMethodProperties;
typedef SPFEECommonTypes::t_MatchProperties t_AuthMethodSearchProperties;

struct t_AuthMethodDesc {
   t_AuthMethod method;
   t_AuthMethodProperties properties;
};

typedef sequence<t_AuthMethodDesc> t_AuthMethodDescList;

exception e_AuthMethodNotSupported {
   // removed t_AuthMethodDescList authMethods;
};

exception e_AccessNotPossible {
};

exception e_AuthenticationError {
   SPFEECommonTypes::Istring sIOR;
};

exception e_AuthMethodPropertiesError {
   SPFEECommonTypes::t_PropertyErrorStruct propertyError;
};

interface i_ProviderInitial {
```

```
// behaviour
// behaviourText
// " A reference to an interface of this type is returned to the PA
// when it has authenticated (or requires no authentication)
// to obtain specific userAgent interfaces.";
// usage
// "to obtain a userAgent reference according to the business
// needs of the consumer";

// requestNamedAccess() types


// Operation 'requestNamedAccess()'
// Used when the user is known to the provider and has already been
// authenticated, either by DPE security or by authenticate()
// input:
// userId: (user name identifying requested user agent.)
//    user_name="anonymous" for anonymous access.
//    user_name may be an empty string, if the provider is
//    using userProperties to identify the          user.
// userProperties: PropertyList which can be used to send
//    additional provider specific user privilege
//    information. This is generic, and can be used to send
//    any type of info to the provider
// output:
// i_nameduaAccess: return: Interface reference of the UserAgent.
// accessSessionId: Identifies the access session the operation is
//    associated with. Must be supplied in all subsequent
//    operations with the InitialAgent and UserAgent.

   void requestNamedAccess (
      in SPFEECommonTypes::t_UserId userId,
      in SPFEECommonTypes::t_UserProperties userProperties,
      out Object  namedAccessIR,           // type: i_ProviderNamedAccess
      out SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out SPFEEAccessCommonTypes::t_AccessSessionId asId
   ) raises (
      e_AccessNotPossible,
      e_AuthenticationError,
      SPFEEAccessCommonTypes::e_UserPropertiesError
   );

// Operation 'requestAnonymousAccess()'
// Used when the user wants to access anonymously to the provider.
// A secure session may already be established by DPE security or by
// authenticate() (the laater does not mean the user is known to the
// provider if a third party authentication protocol is used.)
// input:
// userProperties: may be a null list
// output:  as request_access

   void requestAnonymousAccess (
      in SPFEECommonTypes::t_UserProperties userProperties,
      out Object anonAccessIR, // type: i_ProviderAnonAccess
      out SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out SPFEEAccessCommonTypes::t_AccessSessionId asId
   ) raises (
      e_AccessNotPossible,
      e_AuthenticationError,
      SPFEEAccessCommonTypes::e_UserPropertiesError
   );

}; // i_ProviderInitial
```

```
interface i_ProviderAuthenticate {

// behaviour
// behaviourText
// " A reference to an interface of this type is returned to the PA
// when it wishes to choose this route to authenticate
// itself/mutually to the provider. ";
// usage
// "to agree authentication options supported, acquire
// privelege attributes for the consumer and establish
// an access session between the consumer and the provider";

// getAuthenticationMethods() types


typedef sequence<octet> t_opaque;

//Operations 'getAuthenticationMethods ()'
//input:
// property list used to filter output
//output:
// list of available authentication configurations

   void getAuthenticationMethods (
      in  t_AuthMethodSearchProperties desiredProperties,
      out t_AuthMethodDescList authMethods
   ) raises (
      e_AuthMethodPropertiesError,
      SPFEECommonTypes::e_ListError
   );

// Operation 'authenticate()'
// Used to authenticate a consumer attempting to gain access to a
// user agent. invocation is a prerequsite to establishing client /
// side credentails for establishing secure bindings unless
// an alternative route is used
//input:
// Method: used to identify the authentication method proposed by
//    the client, reflects the composition and generation
//    method of other opaque data
// securityName: name assumed by consumer for authentication. may be
//    null accroding to the authentication method used.
// authenData: opaque data containing consumer attributes to be
//    authenticated
// privAttribReq: opaque specification of the privileges requested
//    by the consumer to create credential for subsequent
//    interactions.
//output:
// privAttrib: privilege attributes returned in response to request.
// continuationData: contains challenge data for the client if the
//    authentication method requires continuation of the
//    protocol
// authSpecificData: data specific to the authentication service
//    used.
// raises:
// e_AuthMethodNotSupported: when the authentication mechanism used
//    by client is not supported by i_iaAuthenticate
```

```
    void authenticate(
        in t_AuthMethod authMethod,
        in string securityName,
        in t_opaque authenData,
        in t_opaque privAttribReq,
        out t_opaque privAttrib,
        out t_opaque continuationData,
        out t_opaque authSpecificData,
        out t_AuthenticationStatus authStatus
    ) raises (
        e_AuthMethodNotSupported
    );

// Operation continue_authentication ()'
// To complete an authentication protocol initiated by authenticate.
// used for second and subsequent continuations.
// input:
// responseData: response from the client to the continuationData
//     output from the to authenticate() or previous calls to
//     continue_authenticate()
// output:
// continuation_data
//     as per authenticate, if continuation is necessary.
// credential_data:
//     as per authenticate, initialiation values or extra
//     items.

    void continueAuthentication(
        in t_opaque responseData,
        out t_opaque privAttrib,
        out t_opaque continuationData,
        out t_opaque authSpecificData,
        out t_AuthenticationStatus authStatus
    );

}; // i_ProviderAuthenticate

};

#endif
```

9.2.4 SPFEEProviderAccess.idl

```
// File SPFEEProviderAccess.idl

#ifndef spfeeprovideraccess_idl
#define spfeeprovideraccess_idl

#include "SPFEECommonTypes.idl"
#include "SPFEEAccessCommonTypes.idl"

#include "SPFEEUserInitial.idl"

module SPFEEProviderAccess {

typedef string t_DateTimeRegistered;     // DRAFT ONLY

struct t_RegisteredInterfaceStruct {
    SPFEECommonTypes::t_InterfaceIndex index;
    SPFEECommonTypes::t_InterfaceStruct interfaceStruct;
        // DateTimeRegistered DRAFT ONLY: need some info on when
        // interface was registered.
    t_DateTimeRegistered when;
    SPFEECommonTypes::t_UserCtxtName where;
};
```

```
typedef sequence<t_RegisteredInterfaceStruct> t_RegisteredInterfaceList;


struct t_UserCtxt {
   SPFEECommonTypes::t_UserCtxtName ctxtName;
   SPFEEAccessCommonTypes::t_AccessSessionId asId;
   Object accessIR;          // type: i_UserAccess
   Object terminalIR;        // type: i_UserTerminal
   Object inviteIR;          // type: i_UserInvite
   Object sessionInfoIR;           // type: i_UserSessionInfo
   SPFEEAccessCommonTypes::t_TerminalConfig terminalConfig;
};


typedef sequence<t_UserCtxt> t_UserCtxtList;

enum t_WhichUserCtxt {
   CurrentUserCtxt,
   SpecifiedUserCtxts,
   AllUserCtxts
};

// Implementation Note:
// Orbix does not allow the creator of a union to set the
// discriminator (switch tag). If true, this union requires
// dummy cases for the other enums of t_WhichUserCtxt.

union t_SpecifiedUserCtxt switch (t_WhichUserCtxt) {
   case SpecifiedUserCtxts: SPFEECommonTypes::t_UserCtxtNameList ctxtNames;
   case CurrentUserCtxt: octet dummy1;
   case AllUserCtxts: octet dummy2;       // value should not be processed
};


typedef SPFEECommonTypes::t_MatchProperties t_DiscoverServiceProperties;
typedef SPFEECommonTypes::t_MatchProperties t_SubscribedServiceProperties;

typedef SPFEECommonTypes::t_MatchProperties t_SessionSearchProperties;
typedef SPFEECommonTypes::t_MatchProperties t_AnnouncementSearchProperties;

enum t_EndAccessSessionOption {
   DefaultEndAccessSessionOption,
   SuspendActiveSessions,
   SuspendMyParticipationActiveSessions,
   EndActiveSessions,
   EndMyParticipationActiveSessions,
   EndAllSessions,
   EndMyParticipationAllSessions
};


typedef SPFEEAccessCommonTypes::t_AppName t_AppName;
typedef SPFEEAccessCommonTypes::t_AppVersion t_AppVersion;
typedef SPFEEAccessCommonTypes::t_AppSerialNum t_AppSerialNum;
typedef SPFEEAccessCommonTypes::t_AppLicenceNum t_AppLicenceNum;

typedef SPFEEAccessCommonTypes::t_ApplicationInfo t_ApplicationInfo;
typedef                 SPFEEAccessCommonTypes::t_StartServiceUAProperties
t_StartServiceUAProperties;
typedef                 SPFEEAccessCommonTypes::t_StartServiceSSProperties
t_StartServiceSSProperties;
```

```
// Exceptions

enum t_SessionErrorCode {
   UnknownSessionError,
   InvalidSessionId,
   SessionDoesNotExist,
   InvalidUserSessionState,
   SessionNotAllowed,
   SessionNotAccepted,
   SessionOpNotSupported
};

exception e_SessionError {
   t_SessionErrorCode errorCode;
   SPFEEAccessCommonTypes::t_UserSessionState state;
};

enum t_UserCtxtErrorCode {
   InvalidUserCtxtName,
   InvalidUserAccessIR,
   InvalidUserTerminalIR,
   InvalidUserInviteIR,
   InvalidTerminalId,
   InvalidTerminalType,
   InvalidNAPId,
   InvalidNAPType,
   InvalidTerminalProperty,
   UserCtxtNotAvailable
};

exception e_UserCtxtError {
   t_UserCtxtErrorCode errorCode;
   SPFEECommonTypes::t_UserCtxtName ctxtName;
   SPFEECommonTypes::t_PropertyErrorStruct propertyError;
      //PropertyError, if errorCode= InvalidTerminalProperty
};

enum t_RegisterUserCtxtErrorCode {
   UnableToRegisterUserCtxt
};

exception e_RegisterUserCtxtError {
   t_RegisterUserCtxtErrorCode errorCode;
};

enum t_EndAccessSessionErrorCode {
   EASE_UnknownError,
   EASE_InvalidOption,
   EASE_ActiveSession,
   EASE_SuspendedSession,
   EASE_SuspendedParticipation
};

exception e_EndAccessSessionError {
   t_EndAccessSessionErrorCode errorCode;
            // sessions causing a problem.
   SPFEECommonTypes::t_SessionIdList sessions;
};
// ExceptionCodes t_ServiceErrorCode;
// Example of Exception codes definition
enum t_ServiceErrorCode {
   InvalidServiceId,
   ServiceUnavailable,
   SessionCreationDenied,
   SessionNotPossibleDueToUserCtxt
   };
```

```
exception e_ServiceError {
   t_ServiceErrorCode errorCode;
};

//  e_StartServiceUAPropertyError & e_StartServiceSSPropertyError
//  are defined to distinguish property errors in
//  t_StartServiceUAProperties & t_StartServiceUAProperties respectively
exception e_StartServiceUAPropertyError {
    // use the errorCodes as for e_PropertyError
    SPFEECommonTypes::t_PropertyErrorStruct propertyError;
};

exception e_StartServiceSSPropertyError {
   // use the errorCodes as for e_PropertyError
   SPFEECommonTypes::t_PropertyErrorStruct propertyError;
};

enum t_ApplicationInfoErrorCode {
   UnknownAppInfoError,
   InvalidApplication,    // Can't use this application with this
          // service/session
   InvalidAppInfo,
   UnknownAppName,                 // I didn't recognise your app name
   InvalidAppName,                 // I don't understand your app name
          // (eg. badly formatted)
   UnknownAppVersion,
   InvalidAppVersion,
   InvalidAppSerialNum,
   InvalidAppLicenceNum,
   AppPropertyError,
   AppSessionInterfacesError,
   AppSessionModelsError,
   AppSIDescError
};

exception e_ApplicationInfoError {
   t_ApplicationInfoErrorCode errorCode;

// t_PropertyErrorStruct:
// Contains the t_PropertyName and t_PropertyValue in error,
// (if t_ApplicationInfoError.errorCode==AppPropertyError
// OR t_PropertyErrorStruct.errorCode==NoPropertyError),
// if error is not due to a property

   SPFEECommonTypes::t_PropertyErrorStruct propertyError;

// t_SessionInterfacesErrorStruct
// Only used if:
// t_AppInfoError.errorCode == AppIntRefInfoError

   SPFEECommonTypes::t_InterfacesErrorStruct itfsError;
};


enum t_AnnouncementErrorCode {
   InvalidAnnouncementId
};

exception e_AnnouncementError {
   t_AnnouncementErrorCode errorCode;
};

interface i_ProviderAccessGetInterfaces {
```

```
// behaviour
// behaviourText
// "This interface allows the user domain to get
// interfaces exported by this provider domain."

// usage
// "This interface is not to be exported across
// Ret RP. It is inherited into the exported interfaces."

   void getInterfaceTypes (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out SPFEECommonTypes::t_InterfaceTypeList itfTypes
   ) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEECommonTypes::e_ListError
   );

   void getInterface (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_InterfaceTypeName itfType,
      in SPFEECommonTypes::t_MatchProperties desiredProperties,
      out SPFEECommonTypes::t_InterfaceStruct itf
   ) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEECommonTypes::e_InterfacesError,
      SPFEECommonTypes::e_PropertyError
   );

   void getInterfaces (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_MatchProperties desiredProperties,
      out SPFEECommonTypes::t_InterfaceList itfs
   ) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEECommonTypes::e_PropertyError,
      SPFEECommonTypes::e_ListError
   );

   }; // i_ProviderAccessGetInterfaces

   interface i_ProviderAccessRegisterInterfaces {

   // behaviour
   // behaviourText
   // "This interface allows the client to register interfaces
   // exported by the client domain."

   // usage
   // "This interface is not to be exported across Ret RP.
   // It is inherited into the exported interfaces."

          // register interfaces to be used only during
          // current access session

      void registerInterface (
         in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
         inout SPFEECommonTypes::t_RegisterInterfaceStruct itf
      ) raises (
         SPFEEAccessCommonTypes::e_AccessError,
         SPFEECommonTypes::e_InterfacesError,
         SPFEECommonTypes::e_RegisterError
      );

      void registerInterfaces (
         in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
```

```
            inout SPFEECommonTypes::t_RegisterInterfaceList itfs
        ) raises (
            SPFEEAccessCommonTypes::e_AccessError,
            SPFEECommonTypes::e_InterfacesError,
            SPFEECommonTypes::e_RegisterError
        );

            // register interfaces which will be accessible
            // outside the access session.

        void registerInterfaceOutsideAccessSession (
            in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
            inout SPFEECommonTypes::t_RegisterInterfaceStruct itf
        ) raises (
            SPFEEAccessCommonTypes::e_AccessError,
            SPFEECommonTypes::e_InterfacesError,
            SPFEECommonTypes::e_RegisterError
        );

        void registerInterfacesOutsideAccessSession (
            in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
            inout SPFEECommonTypes::t_RegisterInterfaceList itfs
        ) raises (
            SPFEEAccessCommonTypes::e_AccessError,
            SPFEECommonTypes::e_InterfacesError,
            SPFEECommonTypes::e_RegisterError
        );

        void listRegisteredInterfaces (
            in SPFEEAccessCommonTypes:: t_AccessSessionSecretId asSecretId,
            in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as,
            out t_RegisteredInterfaceList itfs
        ) raises (
            SPFEEAccessCommonTypes::e_AccessError,
            SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
            SPFEECommonTypes::e_ListError
        );
        void unregisterInterface (
            in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
            in SPFEECommonTypes::t_InterfaceIndex index
        ) raises (
            SPFEEAccessCommonTypes::e_AccessError,
            SPFEECommonTypes::e_InterfacesError,
            SPFEECommonTypes::e_UnregisterError
        );

        void unregisterInterfaces (
            in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
            in SPFEECommonTypes::t_InterfaceIndexList indexes
        ) raises (
            SPFEEAccessCommonTypes::e_AccessError,
            SPFEECommonTypes::e_InterfacesError,
            SPFEECommonTypes::e_UnregisterError
        );

    }; // i_ProviderAccessRegisterInterfaces

    interface i_ProviderAccessInterfaces
              : i_ProviderAccessGetInterfaces,
                i_ProviderAccessRegisterInterfaces
    {
    // behaviour
    // behaviourText
    // "This interface allows the client to get interfaces
```

```
// exported by this domain, and register interfaces exported
// by the client domain."

// usage
// "This interface is not to be exported across Ret RP.
// It is inherited into the exported interfaces."

// No additional operations are defined

}; // i_ProviderAccessInterfaces

interface i_ProviderAccess: i_ProviderAccessInterfaces
{
// behaviour
// behaviourText
// "This interface is the place to put operations which should be
// shared between i_ProviderNamedAccess and i_ProviderAnonAccess.
// Currently none are defined."

// usage
// "This interface is not to be exported across Ret RP.
// It is inherited into the exported interfaces."

// No additional operations are defined

}; // interface i_ProviderAccess


interface i_ProviderNamedAccess
      : i_ProviderAccess
{

      void setUserCtxt (
         in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
         in t_UserCtxt userCtxt
      ) raises (
         SPFEEAccessCommonTypes::e_AccessError,
         e_UserCtxtError
      );

      void getUserCtxt (
         in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
         in SPFEECommonTypes::t_UserCtxtName ctxtName,
         out t_UserCtxt userCtxt
      ) raises (
         SPFEEAccessCommonTypes::e_AccessError,
         e_UserCtxtError
      );

      void getUserCtxts (
         in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
         in t_SpecifiedUserCtxt ctxt,
         out t_UserCtxtList userCtxts
      ) raises (
         SPFEEAccessCommonTypes::e_AccessError,
         e_UserCtxtError,
         SPFEECommonTypes::e_ListError
      );

      void getUserCtxtsAccessSessions (
         in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
         in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as,
         out t_UserCtxtList userCtxts
      ) raises (
```

```
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
    SPFEECommonTypes::e_ListError
);

void registerUserCtxtsAccessSessions (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
    e_RegisterUserCtxtError
);


void listAccessSessions (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    out SPFEEAccessCommonTypes::t_AccessSessionList asList
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEECommonTypes::e_ListError
);

void endAccessSession(
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as,
    in t_EndAccessSessionOption option
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
    e_EndAccessSessionError
);

void getUserInfo(
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    out SPFEEAccessCommonTypes::t_UserInfo userInfo
) raises (
    SPFEEAccessCommonTypes::e_AccessError
);

void listSubscribedServices (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in t_SubscribedServiceProperties desiredProperties,
    out SPFEEAccessCommonTypes::t_ServiceList services
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEECommonTypes::e_PropertyError,
    SPFEECommonTypes::e_ListError
);
void discoverServices(
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in t_DiscoverServiceProperties desiredProperties,
    in unsigned long howMany,
    out SPFEEAccessCommonTypes::t_ServiceList services,
           // type: i_DiscoverServicesIterator
    out Object iteratorIR
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEECommonTypes::e_PropertyError,
    SPFEECommonTypes::e_ListError
);
```

```
void getServiceInfo (
   in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
   in SPFEEAccessCommonTypes::t_ServiceId serviceId,
   in                  SPFEEProviderAccess::t_SubscribedServiceProperties
   desiredProperties,
   out SPFEEAccessCommonTypes::t_ServiceProperties serviceProperties
) raises (
   SPFEEAccessCommonTypes::e_AccessError,
   SPFEEProviderAccess::e_ServiceError
);

void listRequiredServiceComponents (
   in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
   in SPFEEAccessCommonTypes::t_ServiceId serviceId,
   in SPFEEAccessCommonTypes::t_TerminalConfig terminalConfig,
   in SPFEEAccessCommonTypes::t_TerminalInfo terminalInfo,
   // Example of usage for Java applet download:
   // name-value pair describing the url of a Java applet
   // name = "URL"
   // type = "string"
   out SPFEECommonTypes::t_PropertyList locations
) raises (
   SPFEEAccessCommonTypes::e_AccessError,
   SPFEEProviderAccess::e_ServiceError
);

   void listServiceSessions (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEEAccessCommonTypes::t_SpecifiedAccessSession as,
      in t_SessionSearchProperties desiredProperties,
      out SPFEEAccessCommonTypes::t_SessionList sessions
   ) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      SPFEEAccessCommonTypes::e_SpecifiedAccessSessionError,
      SPFEECommonTypes::e_PropertyError,
      SPFEECommonTypes::e_ListError
   );

   void getSessionModels (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_SessionId sessionId,
      out SPFEECommonTypes::t_SessionModelList sessionModels
   ) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      e_SessionError,
      SPFEECommonTypes::e_ListError
   );

   void getSessionInterfaceTypes (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_SessionId sessionId,
      out SPFEECommonTypes::t_InterfaceTypeList itfTypes
   ) raises (
      SPFEEAccessCommonTypes::e_AccessError,
      e_SessionError,
      SPFEECommonTypes::e_ListError
   );

   void getSessionInterface (
      in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in SPFEECommonTypes::t_SessionId sessionId,
      in SPFEECommonTypes::t_InterfaceTypeName itfType,
      out SPFEECommonTypes::t_InterfaceStruct itf
   ) raises (
```

```
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError,
    SPFEECommonTypes::e_InterfacesError
);

void getSessionInterfaces (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId,
    out SPFEECommonTypes::t_InterfaceList itfs
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError,
    SPFEECommonTypes::e_ListError
);

void listSessionInvitations (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    out SPFEEAccessCommonTypes::t_InvitationList invitations
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEECommonTypes::e_ListError
);

void listSessionAnnouncements (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in t_AnnouncementSearchProperties desiredProperties,
    out SPFEECommonTypes::t_AnnouncementList announcements
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    SPFEECommonTypes::e_PropertyError,
    SPFEECommonTypes::e_ListError
);

void startService (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEEAccessCommonTypes::t_ServiceId serviceId,
    in t_ApplicationInfo app,
    in SPFEECommonTypes::t_SessionModelReq sessionModelReq,
    in t_StartServiceUAProperties uaProperties,
    in t_StartServiceSSProperties ssProperties,
    out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_ServiceError,
    e_ApplicationInfoError,
    SPFEECommonTypes::e_SessionModelError,
    e_StartServiceUAPropertyError,
    e_StartServiceSSPropertyError
);

void endSession (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError
);

void endMyParticipation (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError
);
```

```
void suspendSession (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError
);

void suspendMyParticipation (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError
);

void resumeSession (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId,
    in t_ApplicationInfo app,
    out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError,
    e_ApplicationInfoError
);

void resumeMyParticipation (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEECommonTypes::t_SessionId sessionId,
    in t_ApplicationInfo app,
    out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError,
    e_ApplicationInfoError
);

void joinSessionWithInvitation (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEEAccessCommonTypes::t_InvitationId invitationId,
    in t_ApplicationInfo app,
    in SPFEECommonTypes::t_PropertyList joinProperties,
    out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError,
    SPFEEAccessCommonTypes::e_InvitationError,
    e_ApplicationInfoError
);
void joinSessionWithAnnouncement (
    in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
    in SPFEEAccessCommonTypes::t_AnnouncementId announcementId,
    in t_ApplicationInfo app,
    in SPFEECommonTypes::t_PropertyList joinProperties,
    out SPFEEAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
    SPFEEAccessCommonTypes::e_AccessError,
    e_SessionError,
    e_AnnouncementError,
    e_ApplicationInfoError
);
```

```
    void replyToInvitation (
        in SPFEEAccessCommonTypes::t_AccessSessionSecretId asSecretId,
        in SPFEEAccessCommonTypes::t_InvitationId invitationId,
        in SPFEECommonTypes::t_InvitationReply reply
    ) raises (
        SPFEEAccessCommonTypes::e_AccessError,
        SPFEEAccessCommonTypes::e_InvitationError,
        SPFEECommonTypes::e_InvitationReplyError
    );

}; // i_ProviderNamedAccess

interface i_ProviderAnonAccess
       : i_ProviderAccess
{
    // No additional operations defined.

}; // i_ProviderAnonAccess

interface i_DiscoverServicesIterator {

    exception e_UnknownDiscoverServicesMaxLeft {};

    void maxLeft (
        out unsigned long n
    ) raises (
        e_UnknownDiscoverServicesMaxLeft
    );

    // moreLeft: true if there are more interfaces,
    // (accessible thru the iterator (This interface)),
    // false if there are no more interfaces to retrieve

    void nextN (
        in unsigned long n,
             // n >= number returned in services
        out SPFEEAccessCommonTypes::t_ServiceList services,
        out boolean moreLeft
    ) raises (
        SPFEECommonTypes::e_ListError
    );

    void destroy ();

}; // i_DiscoverServicesIterator

}; // module SPFEEProviderAccess

#endif
```

9.3 Ret-RP IDLs

```
// File SPFEERetConsumerInitial.idl
#ifndef spfeeretconsumerinitial_idl
#define spfeeretconsumerinitial_idl
#include "SPFEEUserInitial.idl"

module SPFEERetConsumerInitial {

interface i_ConsumerInitial: SPFEEUserInitial::i_UserInitial {
// No additional operations for Consumer.
};
};
#endif
```

```
// File SPFEERetConsumerAccess.idl
#ifndef spfeeretconsumeraccess_idl
#define spfeeretconsumeraccess_idl
#include "SPFEEUserAccess.idl"

module SPFEERetConsumerAccess {

interface i_ConsumerAccess: SPFEEUserAccess::i_UserAccess {
// No additional operations for Consumer.
}; // i_ConsumerAccess

interface i_ConsumerInvite: SPFEEUserAccess::i_UserInvite {
// No additional operations for Consumer.
}; // i_ConsumerInvite

interface i_ConsumerTerminal: SPFEEUserAccess::i_UserTerminal {
// No additional operations for Consumer.
}; // i_ConsumerTerminal

interface i_ConsumerAccessSessionInfo:
SPFEEUserAccess::i_UserAccessSessionInfo {
// No additional operations for Consumer.
}; // i_ConsumerAccessSessioninfo

interface i_ConsumerSessionInfo: SPFEEUserAccess::i_UserSessionInfo {
// No additional operations for Consumer.
}; // i_ConsumerSessionInfo

};

#endif


// File SPFEERetRetailerInitial.idl
#ifndef spfeeretretailerinitial_idl
#define spfeeretretailerinitial_idl

#include "SPFEEProviderInitial.idl"

module SPFEERetRetailerInitial {

interface i_RetailerInitial: SPFEEProviderInitial::i_ProviderInitial {
// No Retailer specific operations defined.
}; // i_RetailerInitial
interface i_RetailerAuthenticate:
SPFEEProviderInitial::i_ProviderAuthenticate {
// No Retailer specific operations defined.
}; // i_RetailerAuthenticate
};
#endif


// File SPFEERetRetailerAccess.idl
#ifndef spfeeretretaileraccess_idl
#define spfeeretretaileraccess_idl
#include "SPFEEProviderAccess.idl"

module SPFEERetRetailerAccess {

interface i_RetailerAccess {
// behaviour
// behaviourText
// "This interface is the place to put operations which should be
// shared between i_RetailerNamedAccess and i_RetailerAnonAccess.
```

```
// These are specific to the Ret RP, so they don't go in i_ProviderAccess
// Currently none are defined."

// usage
// "This interface is not to be exported across Ret RP.
// It is inherited into the exported interfaces."
// No retailer specific operations are defined
}; // i_RetailerAccess

interface i_RetailerNamedAccess
          :SPFEEProviderAccess::i_ProviderNamedAccess,
          i_RetailerAccess
{
   // Change the name of the interface for Ret RP.
}; // i_RetailerNamedAccess

interface i_RetailerAnonAccess
          : SPFEEProviderAccess::i_ProviderAnonAccess,
          i_RetailerAccess
{
   // Change the name of the interface for Ret RP
}; // i_RetailerAnonAccess
}; // module SPFEERetRetailerAccess
#endif
```

9.4 Ret-RP Subscription IDL Specifications

9.4.1 SPFEESubCommonTypes.idl

```
// File SPFEESubCommonTypes.idl
// Contents:
// This file include common definitions required by the subscription
// magement component and its clients.
// @see SPFEECommonTypes
// @see SPFEEAccessCommonTypes
//

#ifndef SPFEESUBCOMMONTYPES_IDL
#define SPFEESUBCOMMONTYPES_IDL

// This file provides definitions that are common to the service
architecture.
#include "SPFEEAccessCommonTypes.idl"

// Module with common types definitions for subscription management.
module SPFEESubCommonTypes {

// List of Service Identifiers.
typedef sequence<SPFEEAccessCommonTypes::t_ServiceId> t_ServiceIdList;

// Service Types
typedef string t_ServiceType;

// Terminal Type: Just an example.
enum t_TermType {UndefinedTermType,PersonalComputer, WorkStation, TVset,
Videotelephone, Cellularphone, PBX, VideoServer, VideoBridge, Telephone,
G4Fax};

// NAP type: used to determine the instantation of available QoS.
enum t_NapType {UndefinedNapType,NapTypeFixed, NapTypeWireless};

// List of NAPs
typedef sequence<SPFEEAccessCommonTypes::t_NAPId> t_NAPIdList;
```

```
// Terminal presentation technology. This is just an example
enum t_PresentationSupport { UndefinedPresSupp,X11R6, WINDOWS95, MGEG };

// The Account Number represents the Subscriber identifier.
typedef string t_AccountNumber;
typedef sequence<t_AccountNumber> t_SubscriberIdList;

// Types required for SAG management

// Users, terminals and NAPs are considered (subscription) entities
enum t_entityType {user, terminal, nap};

// Entity Id allows to identity uniquely an entity inside the retailer
domain.
union t_entityId switch (t_entityType) {
        case user:     SPFEECommonTypes::t_UserId            userId;
        case terminal: SPFEEAccessCommonTypes::t_TerminalId  terminalId;
        case nap:      SPFEEAccessCommonTypes::t_NAPId       napId;
};
typedef sequence<t_entityId> t_entityIdList;

// An entity is characterized by its identifier and name and a set of
properties.
struct t_Entity {
        t_entityId                        entityId;
        string                            entityName;
        SPFEECommonTypes::t_PropertyList   properties;
};
typedef sequence<t_Entity> t_EntityList;

// The SAE is characterized by an identifier, a name and a set of
properties
struct t_Sae {
        t_entityId       entityId;
        string           entityName;
        SPFEECommonTypes::t_PropertyList   properties; // like password
};

// The SAG identifier identifies a SAG uniquely inside the retailer
domain.
typedef short    t_SagId;
typedef sequence<t_SagId> t_SagIdList;

// A SAG is characterized by its identifier, a textual description of the
// group and the list of entities composing it.
// The identifier is the same as the one for SAG Service profile
corresponding to that SAG.
struct t_Sag {
        t_SagId          sagId;
        string           sagDescription;
        t_entityIdList   entityList;
};
typedef sequence<t_Sag>   t_SagList;

// Subscriber Information:

// Time and Date.
struct t_DateTime {
        string           date;
        string           time;
};

// Textual identification of a person. For example, name, address and
position.
typedef string       t_Person;
```

```
// Indicates the date and time an authorization expires on and the person
who granted it.
struct t_AuthLimit {
        t_DateTime       limitDate;
        string           authority;
}; // note: Shouldn't this be per service contract?

// A subscriber is identified by its account number and characterized by
// name, address, monthly charge, payment record, credit information,
// date which its subscription expires on, the list of subscribed services
// and the list of defined SAGs.
struct t_Subscriber {
        t_AccountNumber           accountNumber;
        SPFEECommonTypes::t_UserId subscriberName;
        t_Person                  identificationInfo;
        t_Person                  billingContactPoint;
        string                    RatePlan;
        any                       paymentRecord;
        any                       credit;
};
typedef sequence<t_Subscriber> t_SubscriberList;

//  This  structure  contains  information  about  the  minimal  required
configuration
// of a service. This is used to specify a configuration for a particular
service session
struct t_RequiredConfiguration {
   t_TermType termType;
   t_NapType nap_type;
   t_PresentationSupport presentation_support;
   SPFEECommonTypes::t_PropertyList others; // to be determined.
};

// Service access rights.
enum t_AccessRight {create,join,be_invited};

// List of possible service access rights.
typedef sequence<t_AccessRight> t_AccessRightList;

// Service Parameter name.
typedef string  t_ParameterName;

// Service Parameter configurability.
enum t_ParameterConfigurability {
    FIXED_BY_PROVIDER, CONFIGURABLE_BY_SUBSCRIBER, CUSTOMIZABLE_BY_USER
};

// Service Parameter value.
typedef any     t_ParameterValue;

// Service Parameters definition:
struct t_Parameter {
    t_ParameterName             name;
    t_ParameterConfigurability  configurability;
    t_ParameterValue            value;
};
typedef sequence<t_Parameter> t_ParameterList;

struct t_ServiceDescription {
        SPFEEAccessCommonTypes::t_ServiceId              serviceId;
        SPFEEAccessCommonTypes::t_UserServiceName        serviceName;
        t_ParameterList                               serviceCommonParams;
        t_ParameterList                              serviceSpecificParams;
};
```

```
// t_serviceTemplate: It describes a service instance.
struct t_ServiceTemplate {
        SPFEEAccessCommonTypes::t_ServiceId         serviceInstanceId;
        SPFEEAccessCommonTypes::t_UserServiceName   serviceInstanceName;
        t_ServiceIdList                            requiredServices;
        t_ServiceDescription                        serviceDescription;
};
typedef sequence<t_ServiceTemplate> t_ServiceTemplateList;

// t_ServiceProfile: It describes a service customization.
typedef string t_ServiceProfileId;
typedef sequence<t_ServiceProfileId> t_ServiceProfileIdList;
struct t_ServiceProfile {
        t_ServiceProfileId                          spId;
        t_ServiceDescription                        serviceDescription;
};
typedef sequence<t_ServiceProfile> t_ServiceProfileList;

// Service Profile for a SAG.
typedef t_ServiceProfile t_SagServiceProfile;

// Service Profile by default in a Service Contract.
typedef t_ServiceProfile t_SubscriptionProfile;

// List of SAG Service Profiles
typedef sequence<t_SagServiceProfile> t_SagServiceProfileList;

// Service Contract: Describes the relationship of a subscriber with the
// provider for the provision of a service.
struct t_ServiceContract {
        SPFEEAccessCommonTypes::t_ServiceId  serviceId;
        t_AccountNumber                     accountNumber;
        short                               maxNumOfServiceProfiles;
        t_DateTime                          actualStart;
        t_DateTime                          requestedStart;
        t_Person                            requester;
        t_Person                            technicalContactPoint;
        t_AuthLimit                         authorityLimit;
        t_SubscriptionProfile               subscriptionProfile;
        t_SagServiceProfileList             sagServiceProfileList;
};

// Notification Type, used in "i_SubscriptionNotify::notify"
enum t_subNotificationType
{NEW_SERVICES,PROFILE_MODIFIED,SERVICES_WITHDRAWN};
// Notification Type, used in "i_ServiceNotify::notify"
enum t_slcmNotificationType
{NEW_SERVICE,TEMPLATE_MODIFIED,SERVICE_WITHDRAWN};
};
#endif // File SPFEESubCommonTypes.idl
```

9.4.2 SPFEERetSubscriberSubscriptionMgmt.idl

```
// File: SPFEERetSubscriberSubscriptionMgmt.idl
// Based on: SPFEEScsSubscriptionService.idl
// Contents: Definition of the online subscription management
// service specific interface.
//

#ifndef SPFEERetSubscriberSubscriptionMgmt_IDL
#define SPFEERetSubscriberSubscriptionMgmt_IDL
#include "SPFEESubCommonTypes.idl"

module SPFEERetSubscriberSubscriptionMgmt {
```

```
interface i_SubscriberSubscriptionMgmt {

enum t_errorType { NameTooLong, AddressTooLong, OtherErrors };
exception e_invalidSubscriberInfo{
      t_errorType errorType;
};
exception e_unknownSubscriber{
      SPFEESubCommonTypes::t_AccountNumber       subscriberId;
};
exception e_applicationError{};
exception e_invalidEntityInfo{};
exception e_unknownSAE{
      SPFEESubCommonTypes::t_entityId    entityId;
};
exception e_unknownSAG{
      SPFEESubCommonTypes::t_SagId       sagId;
};
exception e_invalidSAG{
      SPFEESubCommonTypes::t_SagId       sagId;
};
exception e_invalidContractInfo{};
exception e_invalidSubscriptionProfile{};
exception e_invalidSAGServiceProfile{
          SPFEESubCommonTypes::t_ServiceProfileId     spId;
};
exception e_unknownServiceProfile{
          SPFEESubCommonTypes::t_ServiceProfileId     spId;
};
exception e_unknownServiceId{
      SPFEEAccessCommonTypes::t_ServiceId serviceId;
};
exception e_invalidSearchCriteria{};
exception e_notSubscribedService{
          SPFEEAccessCommonTypes::t_ServiceId serviceId;
};

        // Operations for Subscription and Service Contract handling

        // This operation returns the list of services available for
        subscription and use.
        void listServices (
                out SPFEEAccessCommonTypes::t_ServiceList      serviceList
        ) raises (e_applicationError);
        // This operation creates a subscription for a new customer.
        // The initial list of services the subscriber wants to contract c
        an be specified.
        // It returns:
        // – a unique identifier for the subscriber.
        // – a list of service templates
        void subscribe (
                in SPFEESubCommonTypes::t_Subscriber       subscriberInfo,
                in SPFEESubCommonTypes::t_ServiceIdList       serviceList,
                out SPFEESubCommonTypes::t_AccountNumber     subscriberId,
          out SPFEESubCommonTypes::t_ServiceTemplateList   svcTemplateList
        ) raises (e_invalidSubscriberInfo,
                  e_unknownServiceId,
                  e_applicationError);

        // This operation creates a (set of) new service contract(s) for
        an existing customer.
        // A list of services the subscriber wants to contract is
        specified.
        // It returns a list of service contract management interfaces,
```

```
 //  one for each of the services requested.
 void contractService (
       in SPFEESubCommonTypes::t_ServiceIdList         serviceList,
  out SPFEESubCommonTypes::t_ServiceTemplateList    svcTemplateList
 ) raises (e_unknownSubscriber,
           e_unknownServiceId,
           e_applicationError);

// This operation withdraws a subscription or a list of service
contracts.
 // The list of services the subscriber wants to unsubscribe
 // is an input parameter. If this list is empty, that means
 //  the withdrawal of all the services, and thus the subscription.
 void unsubscribe (
        in SPFEESubCommonTypes::t_ServiceIdList         serviceList
 ) raises (e_unknownSubscriber,
           e_unknownServiceId,
           e_applicationError);

// Operations for Subscriber Information Management.
// -------------------------------------------------------------

// This operation creates a set of entities.
// Sub generates a unique identifier for every entity.
//
 void createSAEs (
         in SPFEESubCommonTypes::t_EntityList     entityList,
         out SPFEESubCommonTypes::t_entityIdList  entityIdList
 ) raises (e_applicationError,
           e_invalidEntityInfo);

 // This operation deletes a set of entities. The entity is
 // removed from all the SAGs it could be assigned to and then
  deleted.
 void deleteSAEs (
         in SPFEESubCommonTypes::t_EntityList     entityList,
         in SPFEESubCommonTypes::t_entityIdList entityIdList
 ) raises (e_applicationError,
           e_unknownSAE);

 // This operation creates a set of SAGs.
 // It returns a set of unique SAG identifiers.
 void createSAGs (
         in SPFEESubCommonTypes::t_SagList    sagList,
         out SPFEESubCommonTypes::t_SagIdList sagIdList
 ) raises (e_applicationError,
           e_invalidSAG,
           e_unknownSAG);

 // This operation deletes a SAG. The entities belonging to that
  SAG are not deleted.
 void deleteSAGs (
         in SPFEESubCommonTypes::t_SagIdList sagIdList
 ) raises (e_applicationError,
           e_unknownSAG);

 // This operation assigns a list of entities to a SAG.
 void assignSAEs (
         in SPFEESubCommonTypes::t_entityIdList entityList,
         in SPFEESubCommonTypes::t_SagId        sagId
 ) raises (e_unknownSAE,
           e_unknownSAG,
           e_applicationError);
```

```
        // This operation removes a list of entities from a SAG.
        void removeSAEs (
                in SPFEESubCommonTypes::t_entityIdList entityList,
                in SPFEESubCommonTypes::t_SagId        sagId
        ) raises (e_unknownSAE,
                  e_unknownSAG,
                  e_applicationError);

        // This operation returns the list of entities assigned to a SAG.
        // If a SAG is not specified, it returns all the entities for that
         subscriber.
        void listSAEs (
                in SPFEESubCommonTypes::t_SagId         sagId,
                out SPFEESubCommonTypes::t_entityIdList entityList
        ) raises (e_unknownSAG,
                  e_applicationError);

        // This operation returns the list of SAGs for that subscriber.
        void listSAGs (
                out SPFEESubCommonTypes::t_SagIdList     sagIdList
        ) raises (e_applicationError);

        // This  operation  returns  the  information  about  a  specific
         subscriber
        void getSubscriberInfo (
                out SPFEESubCommonTypes::t_Subscriber     subscriberInfo
        ) raises (e_applicationError,
                  e_unknownSubscriber);

        // This  operation  modifies  the  information  about  a  specific
         subscriber
       // Only name and address fields are modifiable. The rest are updated
        // only by Sub as a result of other operations -createSAGs,...-
        void setSubscriberInfo (
                in SPFEESubCommonTypes::t_Subscriber     subscriberInfo
        ) raises (e_unknownSubscriber,
                  e_invalidSubscriberInfo,
                  e_applicationError);

        // This operation returns the list of services subscribed by
        //  a specific subscriber.
        //
        void  listSubscribedServices (
               out SPFEEAccessCommonTypes::t_ServiceList      service_list
        ) raises (e_applicationError,
                  e_unknownSubscriber);

       // Operations for Service Contract Management.
       // ------------------------------------------------

        // This operation returns the template for the service.
       void getServiceTemplate (
                in SPFEEAccessCommonTypes::t_ServiceId           serviceId,
                out SPFEESubCommonTypes::t_ServiceTemplate  template
        ) raises (e_applicationError);

        // This operation creates a service contract.
        // This contract can include a set of service profiles.
        void defineServiceContract (
            in SPFEESubCommonTypes::t_ServiceContract     serviceContract,
                out SPFEESubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_invalidContractInfo,
```

```
            e_invalidSubscriptionProfile,
            e_invalidSAGServiceProfile);

    // This operation creates a set of service profiles.
    void defineServiceProfiles (
  in SPFEESubCommonTypes::t_SubscriptionProfile   subscriptionProfile,
  in SPFEESubCommonTypes::t_SagServiceProfileList  sagServiceProfiles,
          out SPFEESubCommonTypes::t_ServiceProfileIdList  spIdList
    )
    ras      e_invalidSubscriptionProfile,
            e_invalidSAGServiceProfile);

    //  his operation  deletes  a  set  of  service  profiles  and  their
     associated SAGs.
    void deleteServiceProfiles (
            in SPFEESubCommonTypes::t_ServiceProfileIdList  spIdList
    )
    raises (e_applicationError,
            e_unknownServiceProfile);

    // This operation returns the list of service profiles identifiers
    void listServiceProfiles (
            in SPFEEAccessCommonTypes::t_ServiceId          serviceId,
            out SPFEESubCommonTypes::t_ServiceProfileIdList  spIdList
    )
    raises (e_applicationError);

    // This operation returns the service contract information.
    // If a (list of) SAG(s) is specified it returns the set of
    // SAG service profile for that(those) SAG(s).
    void getServiceContractInfo (
         in  SPFEEAccessCommonTypes::t_ServiceId           serviceId,
         in  SPFEESubCommonTypes::t_ServiceProfileIdList     spIdList,
         out SPFEESubCommonTypes::t_ServiceContract    serviceContract
    )
    raises (e_applicationError,
            e_unknownServiceProfile);

    // This operation assigns a service profile to a list of SAGs and
     SAEs.
    void assignServiceProfile (
            in SPFEESubCommonTypes::t_ServiceProfileId  spId,
            in SPFEESubCommonTypes::t_SagIdList  sagIdList,
            in SPFEESubCommonTypes::t_entityIdList  saeIdList
    )
    raises (e_applicationError,
            e_unknownSAG,
            e_unknownSAE,
            e_unknownServiceProfile);

    // This operation removes a service profile assignment to a list
   of SAGs and SAEs.
    void removeServiceProfile (
            in SPFEESubCommonTypes::t_ServiceProfileId  spId,
            in SPFEESubCommonTypes::t_SagIdList  sagIdList,
            in SPFEESubCommonTypes::t_entityIdList  saeIdList
    )
    raises (e_applicationError,
            e_unknownSAG,
            e_unknownSAE,
            e_unknownServiceProfile);

    // This operation activates a set of service profiles
    void activateServiceProfiles (
```

```
                in SPFEESubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

        // This operation deactivates a set of service profiles
        void deactivateServiceProfiles (
                in SPFEESubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

}; // i_SubscriberSubscriptionMgmt
};
#endif // SPFEERetSubscriberSubscriptionMgmt_IDL
```

9.4.3 SPFEERetRetailerSubscriptionMgmt.idl

```
#ifndef SPFEERetRetailerSubscriptionMgmt_IDL
#define SPFEERetRetailerSubscriptionMgmt_IDL
#include "SPFEERetSubscriberSubscriptionMgmt.idl"

module SPFEERetRetailerSubscriptionMgmt {

interface i_RetailerSubscriptionMgmt:
SPFEERetSubscriberSubscriptionMgmt::i_SubscriberSubscriptionMgmt
{
// This interface inherits from  i_SubscriberSubscriptionMgmt, and adds a
few
//  operations  for  the  retailer  operator's  access  to  subscription
functionality
// through the Ret-RP.

        // TBD.
        void listSubscribers (
        );

        // TBD.
        void listServiceContracts (
        );

        // TBD.
        void listUsers (
        );

}; // i_RetailerSubscriptionMgmt

};

#endif // SPFEERetRetailerSubscriptionMgmt_IDL
```

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	**Switching and signalling**
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	Languages and general software aspects for telecommunication systems

Geneva, 2001