CORBA Electronic Commerce Domain Specifications

Version 1.0, Month Year

Copyright 1999, Fraunhofer Institut Materialfluss und Logistik
Copyright 1999, Imperial College of Science Technology and Medicine
Copyright 1999, In-Line Software
Copyright 1999, OSM SARL
Copyright 1999, Sprint - Technology Planning and Integration
Copyright 1999, Xerox Corporation

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

PATENT

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

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner. RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMGÆ and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form at *http://www.omg.org/library/issuerpt.htm*.

Contents

1. Overview 1-1

1.1 About the Object Management Group 1-1

1.1.1 What is CORBA? 1-2

1.1.2 What is CORBA E-Commerce? 1-3

1.2 Associated Documents 1-3

1.3 Summary of Key Features 1-4

1.3.1 Session Framework. 1-5

1.3.2 Community Framework 1-5

1.3.3 Collaboration Framework 1-5

1.3.4 DOM Framework 1-5

1.4 Acknowledgments 1-5

2. Session Framework 2-1

2.1 Overview 2-1

2.1.1 Types Derived from the Task/Session Interfaces 2-1

2.1.2 Linkage Types 2-3

2.2 ActiveResource and Associative Interfaces 2-5

2.2.1 ActiveResource. 2-5

2.2.2 Linkage 2-8

2.2.3 Delegation 2-10

2.2.4 Composition 2-11

2.3 ActiveTask and Associative Interfaces 2-11

2.3.1 ActiveTask 2-11

2.3.2 Usage 2-14

2.3.3 Data 2-15

2.3.4 Process 2-15

2.4 Workspace, Desktop, and Containment Associations. 2-15

2.4.1 ActiveWorkspace 2-15

2.4.2 Desktop 2-16

2.4.3 Containment 2-17

2.5 ActiveUser and Supporting Interfaces 2-18

2.5.1 ActiveUser 2-18

2.5.2 LegalEntity 2-19

2.5.3 Jurisdiction 2-20

2.5.4 AbstractTemplate 2-21

2.5.5 SessionFramework IDL 2-23

3. Community Framework 3-1

3.1 Overview 3-1

3.1.1 Object Model 3-3

Contents

3.2 Interfaces 3-3
3.2.1 Membership, Associative, and Qualifying Interfaces 3-3
3.2.2 Member 3-5
3.2.3 Membership 3-6
3.2.4 MembershipKind 3-14
3.3 Community and Derived Interfaces 3-16
3.3.1 Overview 3-16
3.3.2 Community 3-17
3.3.3 Agency 3-18
3.3.4 CommunityFramework IDL 3-18

4. Collaboration Framework 4-1

4.1 Overview 4-1
4.2 Encounter and Associated Interfaces 4-3
4.2.1 Encounter 4-3
4.2.2 Encounter Template 4-6
4.2.3 Implication 4-6
4.3 Collaboration Interfaces 4-8
4.3.1 Collaboration 4-8
4.3.2 CollaborationTemplate 4-15
4.3.3 Trigger 4-18
4.3.4 Command 4-21
4.3.5 Transition 4-21
4.3.6 CompoundTransition 4-22
4.4 Engagement and Associated Interfaces 4-24
4.4.1 Object Model 4-24
4.4.2 EngagementTemplate 4-24
4.4.3 Engagement 4-26
4.4.4 EngagementManifest 4-26
4.5 Voting and Associated Interfaces 4-27
4.5.1 Object Model 4-27
4.5.2 VoteTemplate 4-28
4.5.3 Voting 4-28
4.5.4 VoteManifest 4-29
4.6 Negotiation and Promissory Models 4-30
4.6.1 Bilateral Negotiation 4-30
4.6.2 Multilateral Negotiation 4-34
4.6.3 Promissory Encounter 4-40
4.6.4 CollaborationFramework IDL 4-43

5. DOM Framework 5-1

5.1 Overview 5-1

5.2 DomFramework Wrapper Interfaces 5-1

5.2.1 Extensions 5-3

5.2.2 DomFramework IDL 5-4

5.2.3 DOM Level 1 IDL (errata version) 5-7

Appendix A - Glossary A-1

Appendix B - Object Model B-1

Contents

Overview 1

Contents

This chapter contains the following topics.

Topic	Page
“About the Object Management Group”	1-1
“Associated Documents”	1-3
“Summary of Key Features”	1-4
“Acknowledgments”	1-5

1.1 About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

1.1.1 What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them.

1.1.1.1 CORBA History

CORBA 1.0 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). Included a single language mapping for the C language.

CORBA 1.1 (February 1992) was the first widely published version of the CORBA specification. It closed many ambiguities in the original specification; added interfaces for the Basic Object Adapter and memory management; clarified the Interface Repository, and clarified ambiguities in the object model.

CORBA 1.2 (December 1993) closed several ambiguities, especially in memory management and object reference comparison.

CORBA 2.0 (August 1996) defined true interoperability by specifying how ORBs from different vendors can interoperate. First major overhaul kept the extant CORBA object model, and added several major features:

- dynamic skeleton interface (mirror of dynamic invocation)
- initial reference resolver for client portability
- extensions to the Interface Repository
- "out of the box" interoperability architecture (GIOP, IIOP, DCE CIOP)
- support for layered security and transaction services
- datatype extensions for COBOL, scientific processing, wide characters
- interworking with OLE2/COM

Included in this release were the Interoperability Protocol specification, interface repository improvements, initialization, and two IDL language mappings (C++ and Smalltalk).

CORBA 2.1 (August 1997) added additional security features (secure IIOP and IIOP over SSL), added two language mappings (COBOL and Ada), included interoperability revisions and IDL type extensions.

CORBA 2.2 (February 1998) included the Server Portability enhancements (POA), DCOM Interworking, and the IDL/JAVA language mapping specification.

CORBA 2.3 (June 1999) includes the following new and revised specifications:

- COM/CORBA Part A and B
- ORB Portability IDL/Java
- ORB Interoperability
- Objects by value
- C++ Language Mapping
- IDL to Java Language Mapping
- Java to IDL Language Mapping

CORBA 3.0p (Commercial Release due end of 1999) represents an important specification that adds several major features that are grouped according to Components, Quality of Service, and Java and Internet Integration.

1.1.2 What is CORBA E-Commerce?

There are several specifications that apply to special area markets or domains. Each specialty area represents the needs of an important computing market. The CORBA Electronic Commerce Domain architecture is comprised of specifications that relate to the OMG-compliant interfaces for distributed electronic commerce systems.

In addition to CORBA E-Commerce, other domains include:

CORBA Business

CORBA Finance

CORBA Lifesciences

CORBA Med

CORBA Manufacturing

CORBA Telecoms

CORBA Transportation

As specifications become adopted and approved by OMG, they will be included in the CORBA domain documentation set.

1.2 Associated Documents

The CORBA documentation set includes the following books:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBAfacilities: Common Facilities Architecture and Specification* describes an architecture for Common Facilities. Additionally, it includes specifications based on this architecture that have been adopted and published by the OMG.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820 4300
Fax: +1-508-820 4303
pubs@omg.org
http://www.omg.org

1.3 Summary of Key Features

The CORBA Electronic Commerce Domain architecture is comprised of specifications that relate to the OMG-compliant interfaces for distributed electronic commerce systems. Currently, there are four frameworks established as a result of the Negotiation Facility RFP2. These include the Session Framework, Community Framework, Collaboration Framework, and DomFramework.

The Framework Specification presented under chapters 2 through 5 are targeting potential developers of this facility. Information is presented in the form of a breakdown of modules, interfaces, and types. For each interface, details of attributes, operations, events and additional semantics are provided. The documentation assumes that readers are familiar with the object model defined under the **Task/Session** specification, and have familiarity with the notion of structured events as defined by **CosNotification**.

1.3.1 Session Framework

This chapter covers a set of base interfaces supporting **ActiveUser**, **ActiveTask**, **ActiveWorkspace**, and **ActiveResources**. This module brings together two recently adopted OMG specifications, namely **Task/Session** and **CosNotification**. **Task/Session** specification establishes a framework for people, places and things. The **CosNotification** services are used to extend these definitions with an event model suitable for the electronic commerce domain. Interfaces defined under the **SessionFramework** provide the computational platform for the Community and Collaboration frameworks.

1.3.2 Community Framework

This chapter contains extensions to the **SessionFramework** to support communities of collaborating users and defines the types **Membership**, **Community**, **Agency**, and **Member**.

1.3.3 Collaboration Framework

This chapter contains the definition of **Collaboration**, a process through which different models of collaboration rules can be managed. The **CollaborationFramework** module is defined extensively on interfaces from the **SessionFramework** and **CommunityFramework**.

The specification of three collaborative models cover the following areas:

- **bilateral** negotiation
- **multilateral** negotiation
- **promissory** commitment

1.3.4 DOM Framework

This chapter defines a variant of the **W3C DOM** interfaces that address specific anomalies of the original specification. In particular, the interfaces provide explicit support for OMG language mappings and extensions enabling node identity and access constraint declarations.

1.4 Acknowledgments

The following companies have submitted to or have supported submissions contributing to the CORBA E-Commerce specifications:

- Fraunhofer Institut Materialfluss und Logistik
- Imperial College of Science Technology and Medicine
- In-Line Software
- OSM SARL

- Sprint - Technology Planning and Integration
- Xerox Corporation

Session Framework 2

Contents

This chapter contains the following topics.

Topic	**Page**
“Overview”	2-1
“ActiveResource and Associative Interfaces”	2-5
“ActiveTask and Associative Interfaces”	2-11
“Workspace, Desktop, and Containment Associations”	2-15
“ActiveUser and Supporting Interfaces”	2-18

2.1 Overview

This module defines a set of base interfaces that extend the **Task/Session** framework. Interfaces defined here incorporate an event model based on **CosNotification**, and the addition of operations that extend framework interoperability through the explicit declaration of associations.

2.1.1 Types Derived from the Task/Session Interfaces

SessionFramework provides a set of interfaces that directly extend the Task/Session interfaces to include the formal specification of the structured event produced.

2.1.1.1 Object Model

Table 2-1 Task/Session Derivatives - Interface Summary

Interface	**Description**
ActiveResource	**ActiveResource** is a specialization of **Session::AbstractResource** that includes inheritance from the **CosNotifyComm StructuredPushSupplier** and **StructuredPushConsumer** interfaces. This extension introduces the ability of an **AbstractResource** to expose structured events it is capable of producing and to subscribe to events on a selective basis. Other extensions include operations associated with the binding and release of **Linkage** association.
ActiveTask	**ActiveTask** extends **Session::Task** through the addition of **ActiveResource** and serves as a base type for **Encounter**.
ActiveWorkspace	**ActiveWorkspace** extends **Session::Workspace** through **ActiveResource** and provides a base type for **Community**.
Desktop	**SessionFramework::Desktop** extends **Session::Desktop** and **ActiveWorkspace** defining an event enhanced equivalent of the Task/Session **Desktop**.
ActiveUser	**ActiveUser** extends **Session::User** through the addition of the **CosLifeCycle::FactoryFinder** interface and **LegalEntity**. As a **LegalEntity**, an **ActiveUser** exposes public credentials that may be used under contractual engagement processes.

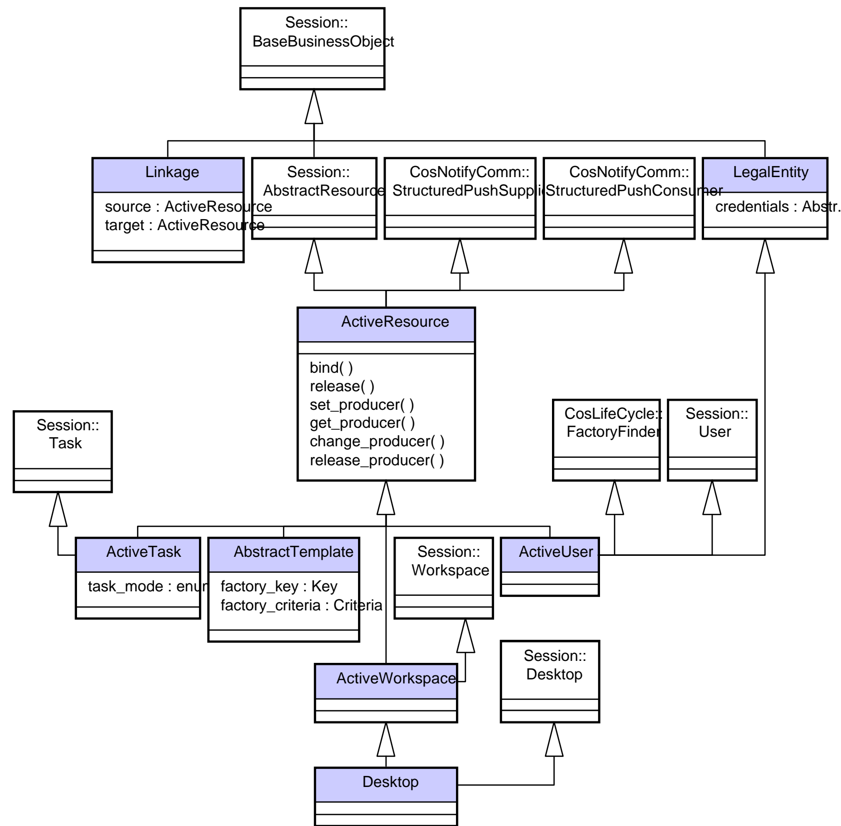

Figure 2-1 Base Task/Session Derivative Interfaces

2.1.2 Linkage Types

A Linkage is a specialization of **Session::BaseBusinessObject** used to describe a generalized relationship between a **source** and a **target** resource. Linkages are used to represent the declaration of concrete relationship types including **Usage**, **Containment**, **Composition, Delegation, Implication**, and **Jurisdiction**.

2.1.2.1 Object Model

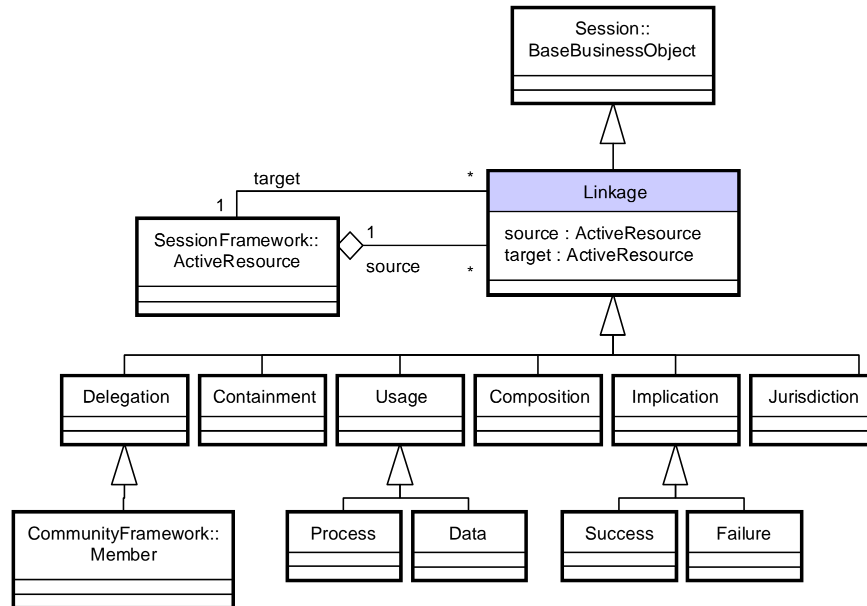

Figure 2–2 Linkage and Derived Types

Table 2-2 Linkage Type Summary

Type	**Source**	**Target**	**Description**
Linkage	ActiveResource	ActiveResource	Abstract base interface that exposes a **source** and **target** of the association.
Delegation	[self reference]	ActiveResource	A role based association that requires a concrete base type that inherits from the target type, and delegates target operation to the target instance.
Containment	ActiveWorkspace	ActiveResource	An association equivalent to the Task/Session **Containment** interface that associates a containing workspace with the contained resource.
Usage	ActiveTask	ActiveResource	An association equivalent to the Task/Session Usage interface that associates a using task with the used resource.
Composition	Composite	ActiveResource	An association that signifies the composition of a target resource within a source composite resource.
Implication	AbstractTemplate	AbstractTemplate	A base type for the **Success** and **Failure** Implication linkage that associates a source template with a target template.
Jurisdiction	ActiveResource	ActiveResource	An association that describes the authority of an **ActiveResource** over another.

Table 2-3 Utility Interface Summary Table

Interface	Description
AbstractTemplate	**AbstractTemplate** is an **ActiveResource** template that exposes a **factory_key** and **criteria**. **AbstractTemplate** is the base type for a set of **EncounterTemplate** types defined under the **CollaborationFramework**.
LegalEntity	A type exposing a set of **AbstractTemplate** instances that defines a **key** and **criteria** for access to public credentials. A **LegalEntity** may be associated to an arbitrary number of **ActiveResource** instances through a **Jurisdiction** linkage.

2.2 ActiveResource and Associative Interfaces

2.2.1 ActiveResource

ActiveResource extends the **Task/Session** specification of **AbstractResource** through the addition of inheritance from the **CosNotifyComm** module **StructuredPushSupplier** and **StructuredPushConsumer** interfaces. Additional operations are included to supporting **Linkage** association and **producer** relationship management. As a structured event supplier, the type exposes lifecycle and feature change events.

2.2.1.1 Object Model

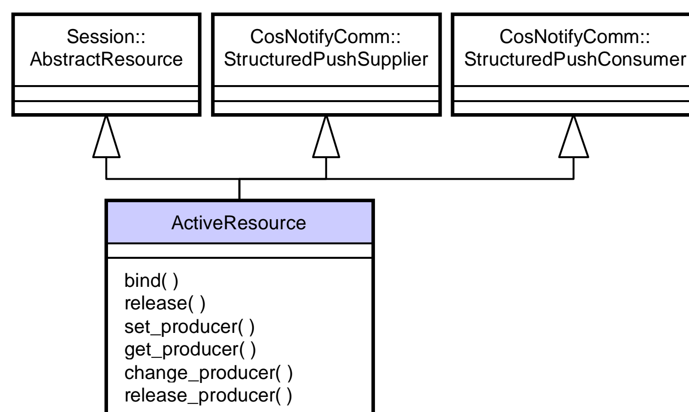

Figure 2–3 ActiveResource Object Model

2.2.1.2 IDL Specification

```
interface ActiveResource :
    Session::AbstractResource,
    CosNotifyComm::StructuredPushSupplier,
    CosNotifyComm::StructuredPushConsumer
{
    exception ResourceUnavailable{ };
    exception ProducerConflict{ };
    void bind(
            in Linkage link
    ) raises (
            ResourceUnavailable
    );
    void release(
            in Linkage link
    );
    ActiveTask get_producer( );
    void set_producer(
            in ActiveTask task
    ) raises (
            ProducerConflict
    );
    void release_producer( );
    void change_producer(
            in SessionFramework::ActiveTask old_task,
            in SessionFramework::ActiveTask new_task
    ) raises (
            ProducerConflict
    );
};
```

2.2.1.3 Linkage Dependencies

ActiveResource extends **AbstractResource** through the addition of operations that support the binding and release of Linkage associations and declaration of producer relationships. Exposure of the bind and release operations ensures that an **ActiveResource** can maintain referential integrity with respect to the **ActiveResource** (see Section 2.2.2, “Linkage,” on page 2-8).

The bind and release operations provide mechanisms through which a binding request can be made to an **ActiveResource** concerning concrete linkage types such as Usage, Containment, or Composition dependency. Both operations take a Linkage as an argument.

```
    void bind(
            in Linkage link
    ) raises (
            ResourceUnavailable
    );
```

```
void release(
        in Linkage link
);
```

2.2.1.4 Produces Relationship

The following IDL provides the interfaces necessary to set, get, and release the reference to the **ActiveTask** producing this resource. The operation **set_producer** associates an **ActiveTask** as the task that is producing the **ActiveResource**. The operation **change_producer** may be used by a mediating client such as **CoordinationFramework::Encounter** to manage production relationships. The **release_producer** enables a task to declare retraction of a producer relationship.

```
void set_producer(
        in ActiveTask task
) raises (
        ProducerConflict
);

void release_producer( );

void change_producer(
        in SessionFramework::ActiveTask old_task,
        in SessionFramework::ActiveTask new_task
) raises (
        ProducerConflict
);
```

2.2.1.5 Structured Events

Under the **CosNotification** specification all events are associated with a unique domain name space. This specification establishes the domain namespace "org.omg.session" for structured events associated with **ActiveResource** and its sub-types.

The **CosNotification** service defines a **StructuredEvent** that provides a framework for the naming of an event and the association of specific properties to that event. All events specified within this facility conform to the **StructuredEvent** interface. This specification requires specific event types to provide the following properties as a part of the **filterable_data** of the structured event header.

Table 2-4 ActiveResource Filterable Data Properties

Name	Type	Description
timestamp	TimeBase::UtcT	Date and time of to which the event is issued.
source	ActiveResource	Active resource raising the event.

Table 2-5 ActiveResource Structured Event Table

Event	Description
update	Notification of the change of a value of an attribute from value x to value y, where x represents the old value and y represents the new value. Supplementary Properties: feature string Attribute name old any Old value new any New value
move	Notification of the transfer of an active resource (move) under which the identity is changed. The source of the event supplies the old instance identity. Supplementary Properties: new ActiveResource Reference containing the new object identity.
remove	Notification of the removal of an ActiveResource
linkage	Notification of the addition or removal of an associated ActiveResource (where association is through a linkage such as Containment, Composition, Usage, Jurisdiction or Delegation). Supplementary Properties: addition boolean True indicates that the Linkage is being added. False indicates the removal of the Linkage.
broadcast	Arbitrary event issued by a client for distribution to all resources associated to the ActiveResource. This event is semantically equivalent to the Task/Session resource_event operation. Supplementary Properties: eventdata any Value to be passed under the event (reference is the Task/Session specification).

2.2.2 *Linkage*

Linkage is a specialization of **Session::BaseBusinessObject** that constitutes an abstract type, which defines a dependency relationship by a source **ActiveResource** towards a dependent target **ActiveResource**. Linkage is the super-type of Usage, Containment, Delegation, Implication, Jurisdiction, and Composition. Instances of Linkage are supplied as arguments to the bind and release operations on **ActiveResource**.

2.2.2.1 Object Model

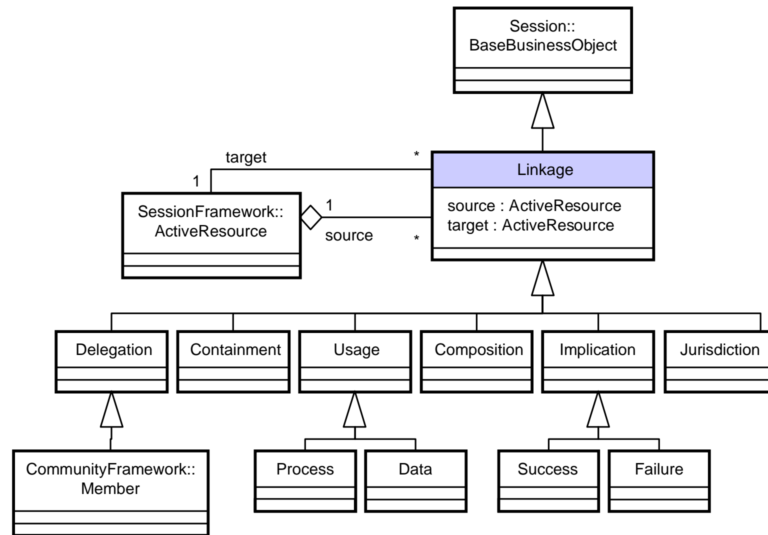

Figure 2–4 Linkage Object Model

2.2.2.2 IDL Specification

```
interface Linkage :
        Session::BaseBusinessObject
        {
        readonly attribute any source;
        readonly attribute any target;
};
```

Table 2-6 Linkage Attribute Table

Name	Type	Properties	Purpose
source	ActiveResource	read-only	Reference to the **ActiveResource** that is requesting or has established a dependency on the **target**.
target	ActiveResource	read-only	Reference to the **ActiveResource** that is the target of a bind operation by source, or maintains a dependency to **source**.

2.2.3 Delegation

Delegation is an abstract specialization of **Linkage** that provides support for the declaration of role based extensions to an **ActiveResource**. A concrete type derived from **Delegation** inherits from the type to which it is associated as **target** and delegates operations to that **target**.

2.2.3.1 Object model

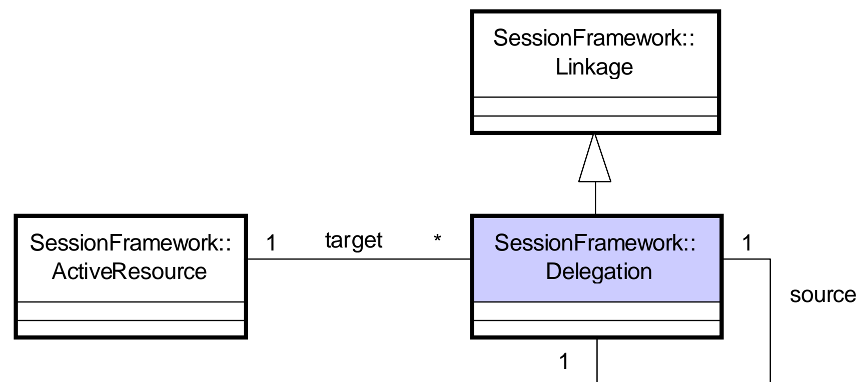

Figure 2–5 Delegation Object Model.

2.2.3.2 IDL Specification

```
interface Delegation :
        Linkage
{
};
```

Table 2-7 Delegation Attribute Table

Name	Type	Properties	Purpose
source	Delegation	read-only	A reference to itself. This may be overridden in a derived type.
target	ActiveResource.	read-only	The resource to which delegation operations will be invoked. A concrete implementation will inherit from the type referenced by **target** and delegates operations to the instance referenced by **target**.

2.2.4 Composition

The Task/Session specification defines **Usage** and **Containment** as mechanisms through which typed relationships among tasks, resources, and workspaces can be expressed.

SessionFramework extends this notion through the addition of the **Composition** relationship type that supports ordered association of composite and composed **ActiveResource** instances. A composite **ActiveResource** is the **source** of the **Composition** linkage. The composed **ActiveResource** is the **target**.

2.2.4.1 Object Model

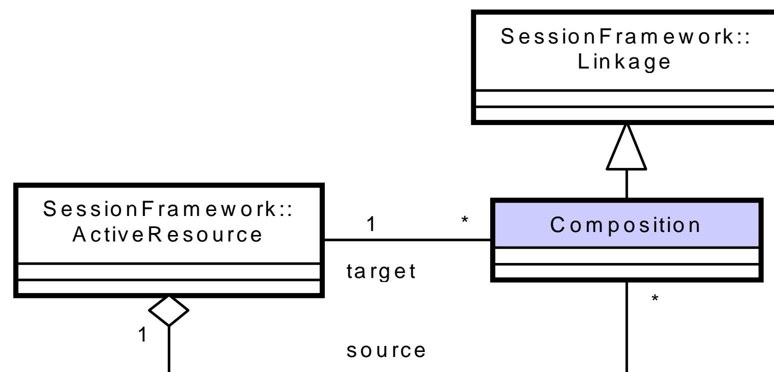

Figure 2–6 Composite Object Model

2.2.4.2 IDL Specification

```
    interface Composition :
    Linkage
    {
};
```

2.3 ActiveTask and Associative Interfaces

2.3.1 ActiveTask

ActiveTask extends the Task/Session specification of **Task** through the addition of **ActiveResource** and the introduction of **task_mode** attribute enabling the exposure of interactive versus batch oriented tasks.

2.3.1.1 Object Model

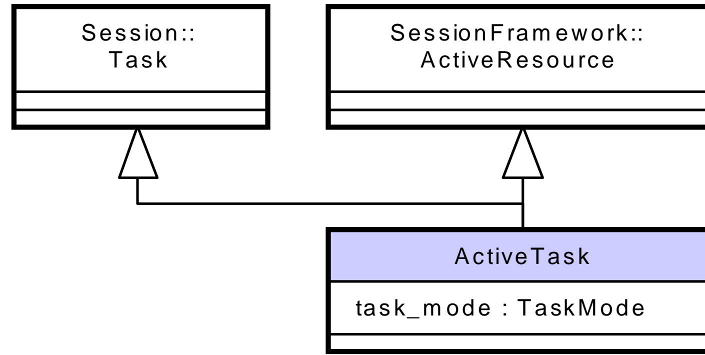

Figure 2–7 ActiveTask Object Model.

2.3.1.2 IDL Specification

```
interface ActiveTask :
      Session::Task,
      ActiveResource
      {
      enum TaskMode{
            BATCH,
            INTERACTIVE
      };
      readonly attribute TaskMode task_mode;
};
```

Table 2-8 Active Task Attribute Table

Name	Type	Properties	Purpose
task_mode	TaskMode	read-only	Indication of the **BATCH** or **INTERACTIVE** mode of execution.

Table 2-9 ActiveTask Structured Event Table

Event	Description
process_state	Notification of the change of state of an ActiveTask. Supplementary properties: value task_state An enumeration as defined by the Task/Session specification.
ownership	Notification of the change of ownership of an ActiveTask. Supplementary properties: owner ActiveUser ActiveUser assigned as owner of the ActiveTask.

2.3.1.3 Resource Usage

Instances of **ActiveTask** are associated to resources they consume through instances of **Usage**. To ensure referential integrity between the task and the resource it consumes, an implementation of **ActiveTask** may request permission to bind to an **ActiveResource** using the **bind** operation, and at a subsequent point, invoke the **release** operation on the target resource.

The **Process** and **Data** usage sub-types differentiate the usage relationship between an **ActiveTask** and a used resource. The **Process** relationship signifies the usage of an **ActiveResource** as the controlling process or editor whereas **Data** signifies a non-process resource.

2.3.1.4 Batch and Interactive Modes

The enumeration **TaskMode** enables a task to be qualified as an interactive or batch oriented task.

```
enum TaskMode{
        BATCH, INTERACTIVE
};
attribute TaskMode task_mode;
```

Table 2-10 TaskMode Enumeration Table

Value	**Description**
BATCH	Indicates that this task is associated to process or editor that will execute and terminate independently of user interaction.
INTERACTIVE	Indicates that this task will invoke an editor that will be manipulated by a user through an interactive interface.

2.3.2 Usage

Usage associates an **ActiveTask** as **source** and an **ActiveResource** as **target**. Cardinality of the Usage relationship is many to many.

2.3.2.1 Object Model

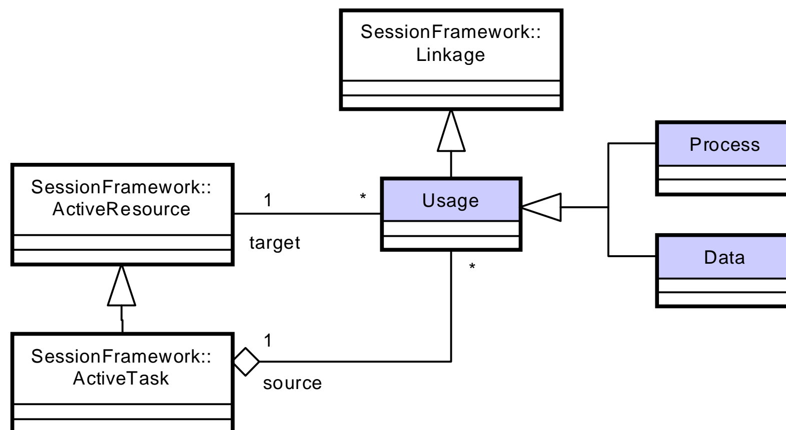

Figure 2–8 .Usage Object Model

2.3.2.2 IDL Specification

```
interface Usage :
Linkage
{
};
```

2.3.3 Data

Data usage differentiates usage as a **target** information **ActiveResource** by a **source ActiveTask** as opposed to Process usage. Cardinality of **Data** is many to many.

2.3.3.1 IDL Specification

```
interface Data :
Usage
{
};
```

2.3.4 Process

Process allows the qualification of usage of a **target ActiveResource** as a process by a **source ActiveTask**. A **source ActiveTask** may have between 0 and 1 instance of **ActiveResource** as a process **target**. An **ActiveResource** may be bound as the **target** under a **Process** linkage by many **ActiveTask** instances.

2.3.4.1 IDL Specification

```
Interface Process :
Usage
{
};
```

2.4 Workspace, Desktop, and Containment Associations

2.4.1 ActiveWorkspace

ActiveWorkspace extends **Session::Workspace** through the addition of **ActiveResource**.

2.4.1.1 Object Model

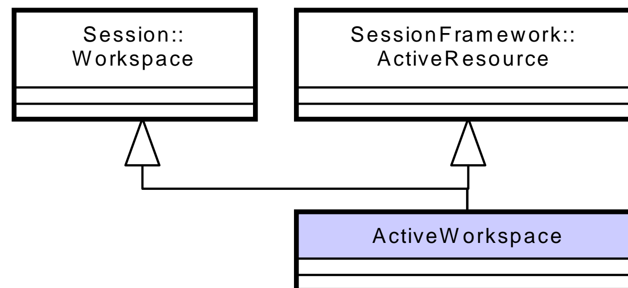

Figure 2–9 ActiveWorkspace Object Model

2.4.1.2 IDL Specification

```
interface ActiveWorkspace :

        Session::Workspace,
        ActiveResource
{
};
```

2.4.1.3 Containment of Resources

ActiveResource containment associations are exposed to the contained resource under the **bind** and **release** operations as instances of **Containment**.

2.4.2 Desktop

Desktop is a specialization of **Session::Desktop** and **ActiveWorkspace**. Inclusion of the desktop interface under this module ensures consistency with respect to event related behavior.

2.4.2.1 Object Model

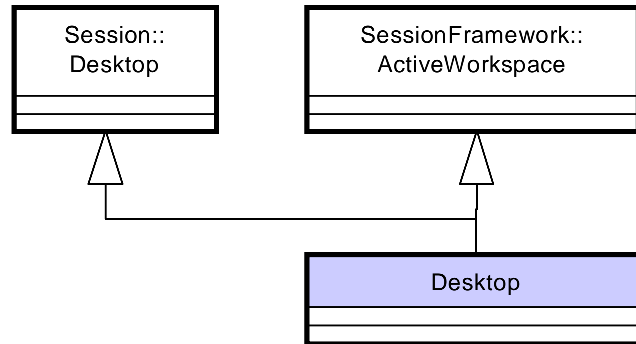

Figure 2–10 Desktop Object Model

2.4.2.2 IDL Specification

```
interface Desktop :
        Session::Desktop,
        ActiveWorkspace
{
};
```

2.4.3 Containment

Containment associates an **ActiveWorkspace** as **source** and an **ActiveResource** as **target**. Cardinality of the **Containment** relationship is many to many.

2.4.3.1 Object Model

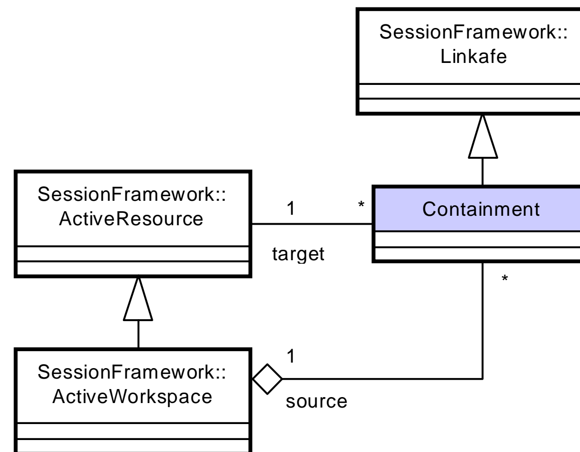

Figure 2–11 Containment Object Model

2.4.3.2 IDL Specification

```
interface Containment :
Linkage
{
};
```

2.5 ActiveUser and Supporting Interfaces

2.5.1 ActiveUser

An **ActiveUser** is an extension of **Session::User**, **ActiveResource**, **LegalEntity** interface, and **CosLifeCycle::FactoryFinder**. **ActiveResource** introduces the consistent behavior concerning events consumption and production, and mechanisms supporting the binding and releasing of linkage associations. **LegalEntity** enables an **ActiveUser** to be associated **[Reviewer: changed association to associated - please verify]** with a **Jurisdiction** relationship relative to **ActiveResource** and exposes public credentials. As a **FactorFinder**, an **ActiveUser** may be used as the **there** argument to a **CosLifeCycle** move or copy operation.

2.5.1.1 Object Model

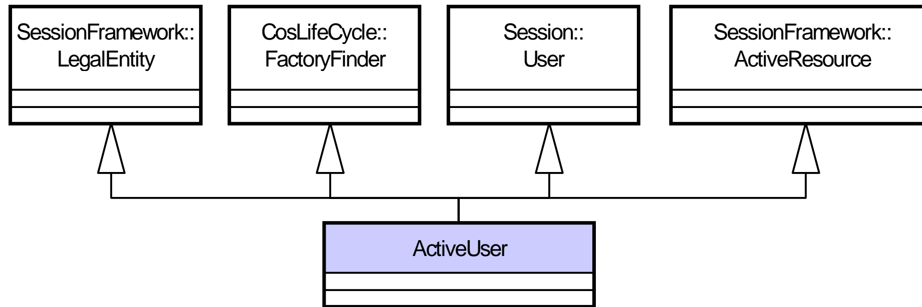

Figure 2–12 Containment Object Model

2.5.1.2 IDL Specification

```
interface ActiveUser :
        Session::User,
        LegalEntity,
        ActiveResource,
        CosLifeCycle::FactoryFinder
        {
};
```

Table 2-11 ActiveUser Structure Event Table

Connected	Optional notification of the success or failure of a task. Supplementary Properties: value boolean True indicates that the user is connected, false indicates that the user is disconnected.

2.5.2 LegalEntity

LegalEntity exposes a sequence of **AbstractTemplate** instances that may be used by a client to construct a create operation against a **GenericFactory**. The structure of credentials and the value of factory key are undefined. Recommendations concerning criteria and factory keys will be provided under subsequent revisions of this specification following the resolution of technology adoption processes dealing with Security interoperability and Public Key Infrastructure services.

2.5.2.1 Object Model

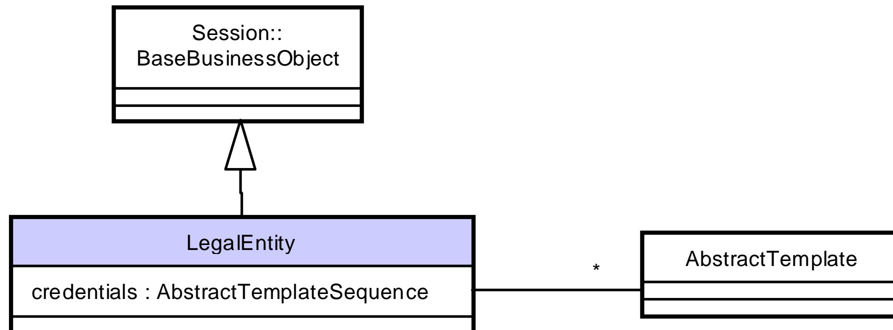

Figure 2–13 LegalEntity Object Model

2.5.2.2 IDL Specification

```
interface LegalEntity :
        Session::BaseBusinessObject
        {
        readonly attribute AbstractTemplateSequence credentials;
};
```

Table 2-12 LegalEntity Attribute Table

Name	Type	Properties	Purpose
credentials	AbstractTemplateSequence	read-only	Used by a client to construct a create operation against a **GenericFactory**.

2.5.3 Jurisdiction

Jurisdiction is a specialization of **Linkage**. Jurisdiction relationships may be used to express hierarchies of authority that client applications may navigate in order to qualify the context of collaboration with respect to the level and scope of authority of respective participants. A **Jurisdiction** linkage associates a source **LegalEntity** with a target **ActiveResource** and implies authority of the **LegalEntity** over the **target** resource.

2.5.3.1 Object Model

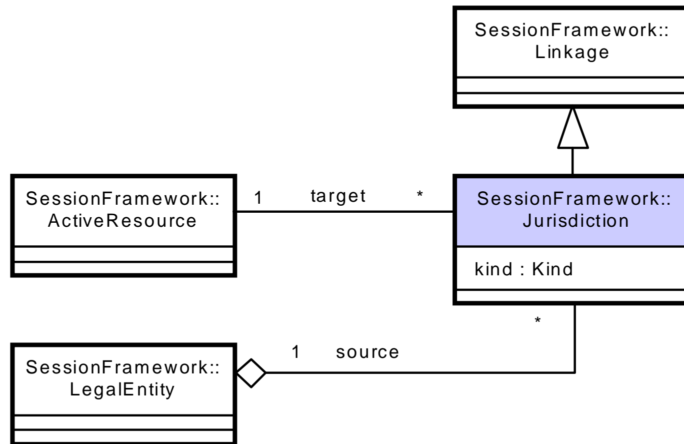

Figure 2–14 Jurisdiction Object Model

2.5.3.2 IDL Specification

```
typedef string Kind;

interface Jurisdiction :
        SessionFramework::Linkage
        {
        readonly attribute SessionFramework::Kind kind;
};
```

Table 2-13 Jurisdiction Attribute Table

Name	Type	Properties	Purpose
kind	Kind	read-only	Application specific string that qualifies the kind of jurisdiction that the relationship infers.

2.5.4 AbstractTemplate

AbstractTemplate is a type that exposes a **factory_key** and **factory_criteria** used by clients under operations dealing with **CosLifeCycle** factory services. **AbstractTemplate** is a base type for the **CollaborationFramework** interface **EncounterTemplate** and the **CommunityFramework** interface **MembershipKind**.

2.5.4.1 *Object Model*

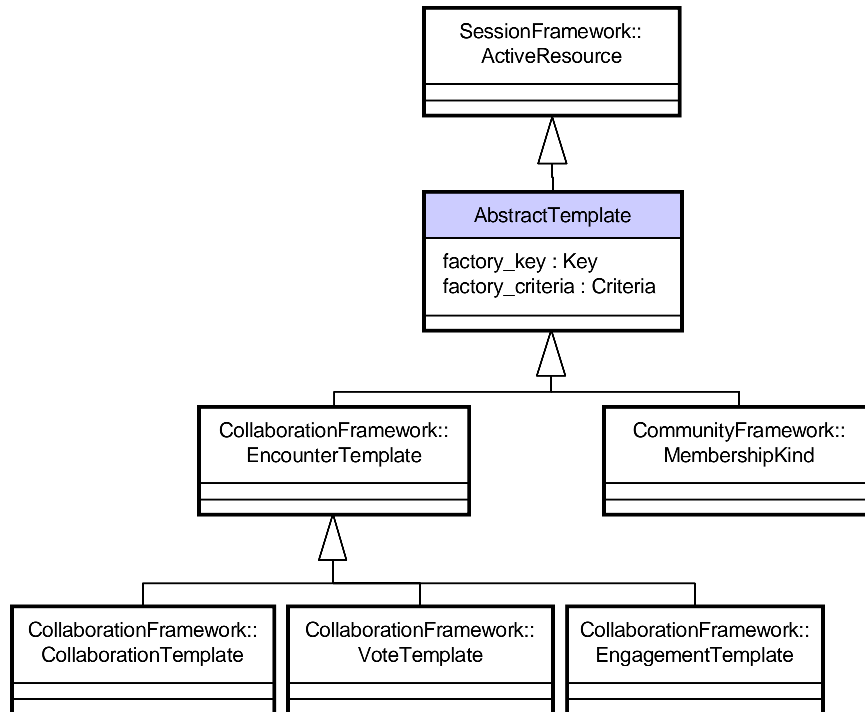

Figure 2–15 AbstractTemplate Object Model

2.5.4.2 *IDL Specification*

```
interface AbstractTemplate :
        ActiveResource
{
        readonly attribute CosLifeCycle::Key factory_key;
        readonly attribute CosLifeCycle::Criteria factory_criteria;
};
```

Table 2-14 AbstractTemplate Attribute Table

Name	Type	Properties	Purpose
factory_key	CosLifeCycle::Key	read-only	Argument to CosLifeCycle factory finder that identifies the type of factory to find.
factory_criteria	CosLifeCycle::Criteria	read-only	Argument to a CosLifeCycle generic factory.

2.5.5 SessionFramework IDL

```
// file SessionFramework.idl

#ifndef _SESSION_FRAMEWORK_IDL_
#define _SESSION_FRAMEWORK_IDL_
#pragma prefix "omg.org"

#include <Session.idl>
#include <CosNotifyComm.idl>
#include <CosPropertyService.idl>
#include <TimeBase.idl>

module SessionFramework{

        // forward declarations

        interface ActiveResource;
        interface ActiveTask;
        interface ActiveWorkspace;
        interface Desktop;
        interface LegalEntity;
        interface ActiveUser;

        interface Linkage;
        interface Usage;
        interface Containment;
        interface Delegation;
        interface Jurisdiction;
        interface Composition;

        interface AbstractTemplate;

        // typedefs

        typedef string Kind;
        typedef sequence <ActiveResource> ActiveResourceSequence;
        typedef sequence <AbstractTemplate> AbstractTemplateSequence;
        typedef sequence <ActiveUser> ActiveUserSequence;
        typedef sequence <ActiveTask> ActiveTaskSequence;
        typedef sequence <ActiveWorkspace> ActiveWorkspaceSequence;

        // iterators

        interface ActiveResourceIterator : CosCollection::Iterator{};
        interface AbstractTemplateIterator: CosCollection::Iterator{};
        interface ActiveUserIterator : CosCollection::Iterator{};
        interface ActiveTaskIterator : CosCollection::Iterator{};
        interface ActiveWorkspaceIterator : CosCollection::Iterator{};
```

```
    // base types

    interface ActiveResource :
        Session::AbstractResource,
        CosNotifyComm::StructuredPushSupplier,
        CosNotifyComm::StructuredPushConsumer {

        exception ResourceUnavailable{ };
        exception ProducerConflict{ };

        void bind(
            in Linkage link
        ) raises (
            ResourceUnavailable
        );
        void release(
            in Linkage link
        );

        // setting, getting and releasing a producer

        ActiveTask get_producer( );
        void set_producer(
            in ActiveTask task
        ) raises (
            ProducerConflict
        );
        void release_producer( );
        void change_producer(
                in SessionFramework::ActiveTask old_task,
                in SessionFramework::ActiveTask new_task
        ) raises (
            ProducerConflict
        );
};

interface ActiveTask :
        Session::Task,
        ActiveResource
        {
        enum TaskMode{
                BATCH,
                INTERACTIVE
        };
        readonly attribute TaskMode task_mode;
};

interface ActiveWorkspace :
        Session::Workspace,
        ActiveResource {
};
```

```
interface Desktop :
        Session::Desktop,
        ActiveResource {
};

// ActiveUser

interface LegalEntity :
        Session::BaseBusinessObject {
        readonly attribute AbstractTemplateSequence credentials;
};

interface ActiveUser :
        Session::User,
        LegalEntity,
        ActiveResource,
        CosLifeCycle::FactoryFinder
        {
};

// Extensions

interface Linkage :
        Session::BaseBusinessObject {
        readonly attribute any source;
        readonly attribute any target;


interface Delegation :
        Linkage {
};

interface Usage : Linkage{ };
interface Data : Usage{ };
interface Process : Usage{ };
interface Containment : Linkage{ };
interface Composition : Linkage{ };

interface Jurisdiction :
        SessionFramework::Linkage {
        readonly attribute SessionFramework::Kind kind;
};

// templates

interface AbstractTemplate :
        ActiveResource {
        readonly attribute CosLifeCycle::Key factory_key;
        readonly attribute CosLifeCycle::Criteria factory_criteria;
};
```

```
}; // end SessionFramework Module

#endif // _SESSION_FRAMEWORK_IDL_
```

Community Framework 3

Contents

This chapter contains the following topics.

Topic	Page
"Overview"	3-1
"Interfaces"	3-3
"Community and Derived Interfaces"	3-16

3.1 Overview

Interfaces defined under the Community module fall into two categories:

1. Interfaces supporting membership management.
2. Interfaces defining Community, the derived interface Agency.

Table 3-1 CommunityFramework Interface Summary Table

Interface	Description
Membership	A specialization of **ActiveResource** that enables association of instances of the type **Member** in accordance with rules exposed under a **MembershipKind**. A **Membership** exposes interfaces through which **Member** instances may be added, removed, and listed relative to the kind of participation exposed by a **MembershipKind** hierarchy.

Table 3-1 CommunityFramework Interface Summary Table

MembershipKind	Definition of constraints for a given **MembershipKind**. Constraints include the maximum number of members that may be associated under the kind, quorum value indicating the number of members that kind must be associated and connected before the Member is considered valid, privacy policy declarations, and policies concerning the semantics of membership hierarchy.
Member	A role of **ActiveUser**, defined as a specialization of **Linkage** that associates a target **ActiveUser** with a **Membership**. As a Membership may be a hierarchy of Membership instances, an instance of Member may be associated as a member at many levels within the hierarchy.
Community	A specialization of **ActiveWorkspace**, **Membership**, and **FactoryFinder**. As an **ActiveWorkspace**, a Community is a place containing **ActiveResources**. As a **Membership**, a **Community** exposes policy concerning membership and the association of Member kind hierarchies. As a **FactoryFinder**, **Community** represents a possible target under a **copy** or **move** operation.
Agency	A specialization of **Community** and **LegalEntity** that introduces the notion of legal community such as a company that maintains jurisdiction of a set of resources. Agency, through **LegalEntity** and **Jurisdiction**, enables the qualification of the authority of a **Member** within a negotiation or other collaborative encounter.

3.1.1 Object Model

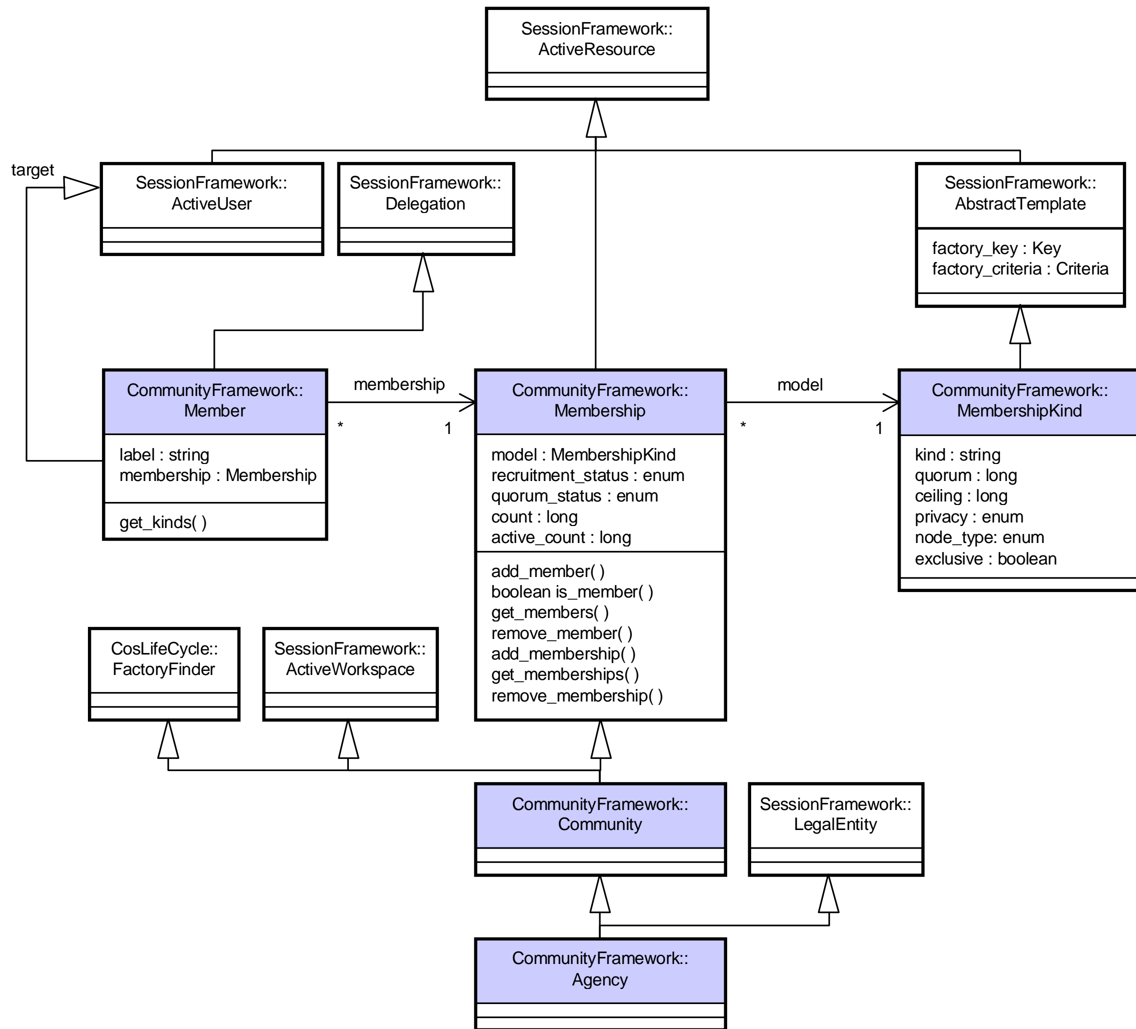

Figure 3-1 CommunityFramework Object Model

3.2 Interfaces

3.2.1 Membership, Associative, and Qualifying Interfaces

Membership is a base type to **Community** and **Encounter**. A **Community** constitutes a set of artifacts shared by the community members. An **Encounter** is a collaborative process involving a set of members. A **Member** is a role of a user associated to an instance of **Membership**. A **Membership** is the run-time

instantiation of a **MembershipKind** hierarchy that defines the kind of memberships that may be attributed to **Members** within the **Membership**. As such, the role of a user is a function of the **Membership** to which a **Member** is associated.

Instances of both **MembershipKind** and **Membership** are associated to subsidiary instances through composition relationships. **Composition** relationships between **MembershipKind** define the hierarchy of roles supported by a single **Membership** instance. Composition relationships between Membership instances define process centric hierarchies, such as a parent and subsidiary negotiation.

3.2.1.1 Object Mode

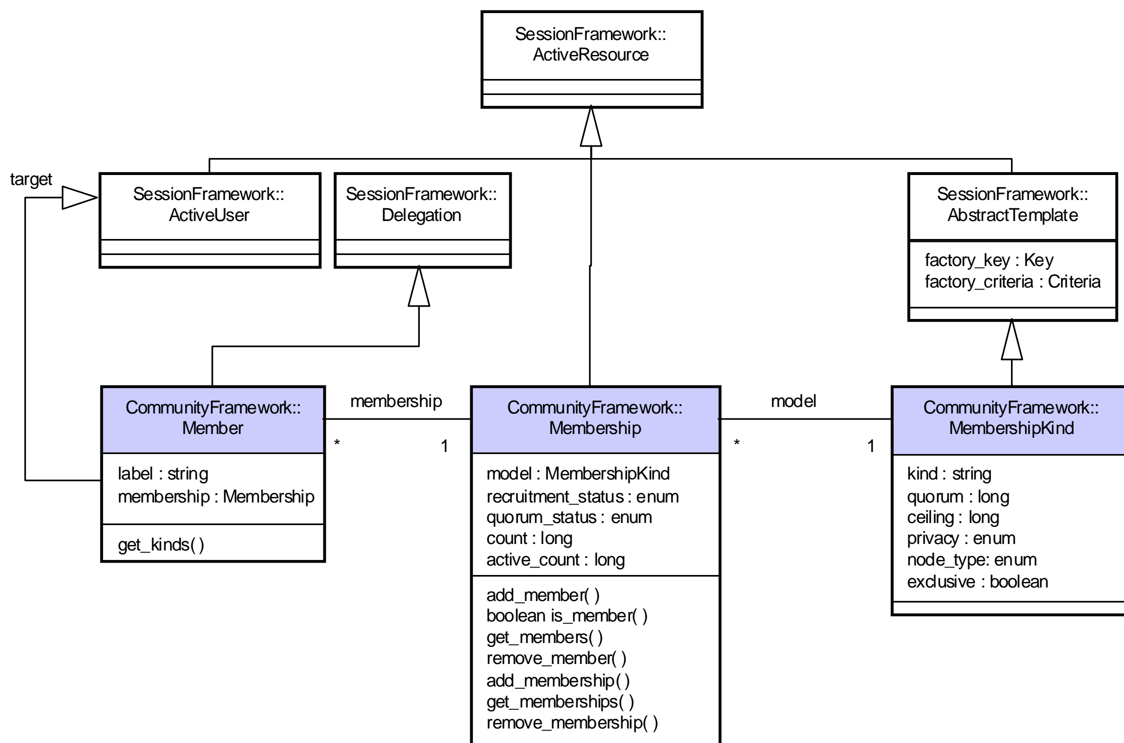

Figure 3-2 Member, Membership and MembershipKind Object Model

3.2.1.2 Example

The following (non-UML) illustration depicts an example of a **Membership**, its purpose in associating the set of **Member** instances Michael, Carol, Alice, and Bob, and the structure of membership kinds (**MembershipKind** instances) that each user is associated with under the **Membership**. In this example, Michael, Carol, and Alice are associated with the membership kinds **consumer** and by virtue of being a

consumer are also associated to the role of **participant**. Alice is both **consumer** and **customer**. As a **customer** she is also a **signatory**. Bob participates to the Membership as **provider**, **signatory**, and **participant**.

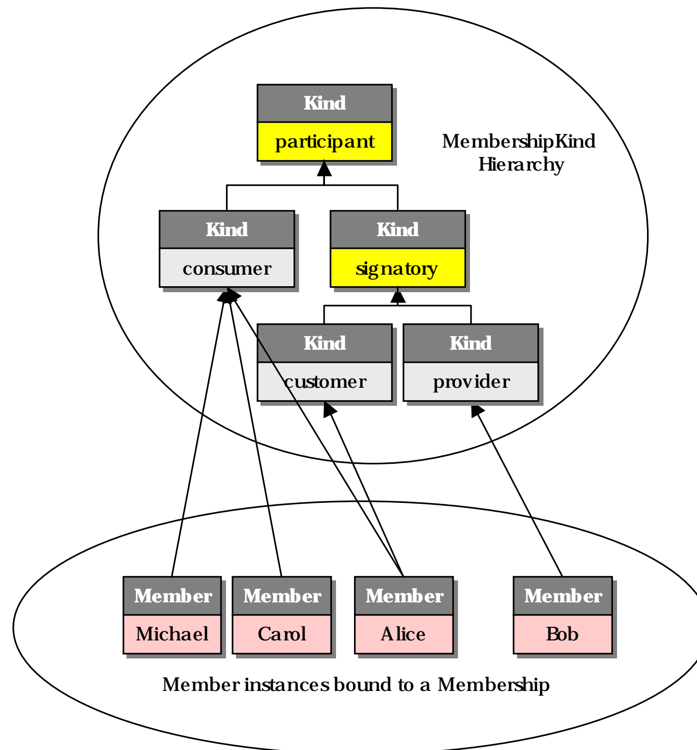

Figure 3-3 Schematic Example of a Membership

The **Member** type manages the state and association of a **Membership** and mediates between a set of **Member** instances and a stateless **MembershipKind** hierarchy. **Member** instances are associated to a **MembershipKind** through operations exposed on the **Membership** interface.

3.2.2 Member

A **Member** represents the participation of an **ActiveUser** to an instance of **Membership**. The association to **ActiveUser** is achieved through a reference to the **ActiveUser** under the **target** attribute inherited from the **Delegation** interface. The attribute **label** may be used as a preferred name of the user relative to other members of the **Membership**. The operation **list_kinds** returns the set of **MembershipKind** instances to which the **Member** is associated within the scope of the **Membership**.

3.2.2.1 IDL Specification

```
interface Member :
    SessionFramework::Delegation,
    SessionFramework::ActiveUser
    {
    attribute string label;
    readonly attribute CommunityFramework::Membership membership;
    void get_kinds(
        out MembershipKindSequence kind_list,
        out MembershipKindIterator kind_iterator
    ) raises (
        PrivacyConflict
    );
};
```

Table 3-2 Member Attribute Table

Name	Type	Properties	Purpose
label	long		The name by which a Member is known to the Community.
membership	Membership	read-only	The root Membership to which this instance of Member is associated.

3.2.2.2 Listing Kind Attributed to a Member

The **get_kinds** operation enables clients to invoke requests against a **Member** to retrieve reference to the **MembershipKind** instances that defines the kind of memberships that the user is associated with within the scope of the **Membership**.

```
void get_kinds(
        out MembershipKindSequence kind_list,
        out MembershipKindIterator kind_iterator
) raises (
        PrivacyConflict
);
```

3.2.3 Membership

A **Membership** is an interface that corresponds to the run-time creation of an association between zero to many **Member** instances and a root **MembershipKind**, and between 0 and many subsidiary **Membership** instances. The **Membership** type exposes operations enabling the addition, listing, and removing of members, querying individual **Member** participation, and features exposing the state of the **Membership**.

3.2.3.1 IDL Specification

```
interface Membership :
    SessionFramework::ActiveResource
{
    readonly attribute MembershipKind model;
    enum RecruitmentStatus{
            OPEN_MEMBERSHIP,
            SUSPENDED_MEMBERSHIP,
            CLOSED_MEMBERSHIP
    };
    readonly attribute RecruitmentStatus recruitment_status;
    exception RecruitmentConflict{
            RecruitmentStatus reason;
    };
    enum QuorumStatus {
            QUORUM_REACHED,
            QUORUM_PENDING,
            QUORUM_UNREACHABLE
    };
    readonly attribute QuorumStatus quorum_status;
    readonly attribute long count;
    readonly attribute long active_count;
    exception AttemptedCeilingViolation{ };
    exception AttemptedExclusivityViolation{ };
    exception VirtualKind{ };
    exception UnknownKind{ };
    exception MembershipRejected{
            Membership source;
            string reason;
    };
    Member add_member(
            in SessionFramework::ActiveUser user,
            in CommunityFramework::MembershipKind kind
    ) raises (
            AttemptedCeilingViolation,
            AttemptedExclusivityViolation,
            RecruitmentConflict,
            MembershipRejected,
            UnknownKind,
            VirtualKind
    );
    boolean is_member(
            in CommunityFramework::Member member,
            in CommunityFramework::MembershipKind kind
    ) raises (
            PrivacyConflict
    );
    void get_members(
            in MembershipKind kind,
```

```
                out MemberSequence member_list,
                out MemberIterator member_iterator
        ) raises (
                PrivacyConflict
        );
        void remove_member(
                in CommunityFramework::Member member
        );
        void add_membership(
                in CommunityFramework::Membership membership
        );
        void get_memberships(
                out MembershipSequence membership_list,
                out MembershipIterator membership_iterator
        ) raises (
                PrivacyConflict
        );
        void remove_membership(
                in CommunityFramework::Membership membership
        );
};
```

Table 3-3 Membership Attribute Table

Name	Type	Properties	Purpose
model	MembershipKind	read-only	Template that defines constraints associated to and enforced by an instance of **Membership**.
recruitment_status	RecruitmentStatus	read-only	Refer to Section 3.2.3.5, "Recruitment Status," on page 3-10.
quorum_status	QuorumStatus	read-only	Refer to Section 3.2.3.6, "Quorum Status," on page 3-11.
count	long	read-only	The number of **Member** instances associated with the **Membership**.
active_count	long	read-only	The number of **Members** associated to the **Membership** and connected.

3.2.3.2 Membership Semantics

Association of a **Member** to a **MembershipKind** grants that user a role within the membership qualified by the **MembershipKind** kind and template parameters. Where a **MembershipKind** is subsidiary to another **MembershipKind**, the **Member** associated to the subsidiary is implicitly considered to inherit the membership kind of the parent **MembershipKind** (refer to earlier example).

MembershipKind exposes the enumeration **VIRTUAL_NODE** and **PHYSICAL_NODE** under the attribute **node_kind**. Virtual nodes provide a useful mechanism for aggregating membership kinds but does not directly support the association of **Members**. Instead, subsidiary associations infer association to a virtual **MembershipKind**.

Where a **PHYSICAL_KIND MembershipKind** is a parent to another **PHYSICAL_KIND**, the removal of a **Member** association from the parent implies removal of the member from all subsidiary kinds.

3.2.3.3 Member Addition

Instances of **Member** may be added to a **Membership** using the **add_member** operation. The **add_member** operation takes an **ActiveUser** as argument identifying the user to be bound to the **Membership**, and a reference to a **MembershipKind** under the **kind** argument. Where a user is already a member of a **Membership**, and the **add_user** operation is invoked in order to supplement the **MembershipKind** associations, the **add_user** operation will return the same **Member** instance.

```
Member add_member(
        in SessionFramework::ActiveUser user,
        in CommunityFramework::MembershipKind kind
) raises (
        AttemptedCeilingViolation,
        AttemptedExclusivityViolation,
        RecruitmentConflict,
        MembershipRejected,
        UnknownKind,
        VirtualKind
);
```

An attempt to add a member when the value of ceiling (exposed under the **MembershipKind**) is greater than or equal to count will result in an **AttemptedCeilingViolation**. An attempt to add a Member, representing an **ActiveUser** that is already represented within a **Membership** under an existing **Member** instance, while a **MembershipKind** value of **exclusive** is true will result in the raising of an AttemptedExclusivityViolation exception. An attempt to add a Member while **recruitment_status** is **CLOSED_MEMBERSHIP** will cause the RectrutmentConflict exception to be raised. An attempt to reference a **MembershipKind** under the **kind** argument that unknown with the scope of the **Membership** model **MembershipKind** will cause the raising of the UnknownKind exception. An attempt to add a user to **MembershipKind** exposing the **VIRTUAL_NODE** as the value of **node_descriptor** will cause the raising of the VirtualKind exception.

```
exception RecruitmentConflict{
        RecruitmentStatus reason;
};
exception AttemptedCeilingViolation{ };
exception AttemptedExclusivityViolation{ };
```

```
exception VirtualKind{ };
exception UnknownKind{ };
exception MembershipRejected{
        Membership source;
        string reason;
};
```

3.2.3.4 Member Removal

Membership removal is invoked using the **remove_member** operation. An implementation of **MembershipDomain** is required to notify the removal of a Member from a domain through the **removal_notification** operation on the instance of **Member** being removed.

```
void remove_member(
    in CommunityFramework::Member member
);
```

On addition or removal of a Member from the domain, an implementation of **Membership** is required to increment or decrement respectively the value of the **count** and **active_count** attributes and signal a change notification event. Changes to the connected status of a **Member** are also reflected in the **active_count** attribute. The **active_count** corresponds to the number of Member instances that are connected (refer **Session::User, connect_state**).

```
readonly attribute long count;
readonly attribute long active_count;
```

3.2.3.5 Recruitment Status

The status of a **Membership** instance is exposed through the **recruitment_status** and **quorum_status** attribute values. The **recruitment_status** attribute exposes a value of **OPEN_MEMBERSHIP**, **SUSPENDED_MEMBERSHIP**, and **CLOSED_MEMBERSHIP** that control the behavior of the **add_member** and **remove_member** operations. Under a closed membership, addition or removal of members is disabled. Under a suspended membership, the addition and removal operations may be invoked; however, an implementation may delay the registration of the **Member** up to the point that the **Membership** is re-opened.

```
enum RecruitmentStatus{
        OPEN_MEMBERSHIP,
        SUSPENDED_MEMBERSHIP,
        CLOSED_MEMBERSHIP
};

readonly attribute RecruitmentStatus recruitment_status;
```

Table 3-4 RecruitmentStatus Enumeration Table

Value	Description
OPEN_MEMBERSHIP	Invocation of the **add_member** and **remove_member** operations is enabled.
SUSPENDED_MEMBERSHIP	Invocation of the **add_member** and **remove_member** operations is enabled; however, an instance of **Membership** may not consider the **Member** association as valid (as exposed by the **is_member** and **get_members** operations).
CLOSED_MEMBERSHIP	Invocation of the **add_member** and **remove_member** operations is disabled.

3.2.3.6 Quorum Status

Membership quorum level (exposed under the **MembershipKind**) defines the number of **Member** instances that are required for the **Membership** to be considered valid. For example, a bilateral negotiation requires a quorum of 2. Prior to reaching **quorum** (**count** is less than **quorum**) the value of **quorum_status** is **QUORUM_PENDING**. On reaching **quorum**, the **quorum_status** is **QUORUM_REACHED**. If the value of **ceiling** is less than **quorum**, **QUORUM_UNREACHABLE** will be exposed. Both **quorum** and **ceiling** are features exposed by the **MembershipKind** referenced by the **model** attribute.

```
enum QuorumStatus {
    QUORUM_REACHED,
    QUORUM_PENDING,
    QUORUM_UNREACHABLE
};

readonly attribute QuorumStatus quorum_status;
```

Table 3-5 QuarumStatus Enumeration Table

Value	Description
QUORUM_REACHED	The number of **Member** instances associated with **MembershipKind** is equal to or exceeds the **MembershipKind quorum** level required.
QUORUM_PENDING	The number of **Member** instances associated with **MembershipKind** is less than the **MembershipKind quorum** level required.
QUORUM_UNREACHABLE	The **MembershipKind quorum** level required is greater than the **ceiling** and as such **quorum** of the **Membership** cannot be achieved.

3.2.3.7 Membership Disclosure Operations

The **Membership** type provides a number of operations enabling navigation of the **Membership** structure and access to **Member** kind associations. The **is_member** operation enables a client to query if an instance of **Member** is recognized by the **Membership** as associated to a particular kind within the scope of applicable privacy restrictions.

```
boolean is_member(
        in CommunityFramework::Member member,
        in CommunityFramework::MembershipKind kind
) raises (
        PrivacyConflict
);
```

The **get_members** operation returns all members of a **Membership** holding the **MembershipKind** passed in under the **kind** argument within the restrictions of the applicable privacy policy.

```
void get_members(
        in MembershipKind kind,
        out MemberSequence member_list,
        out MemberIterator member_iterator
) raises (
        PrivacyConflict
);
```

3.2.3.8 Structural Operations

The **add_membership** operation enables a client application to introduce a subsidiary membership to an existing membership. This operation is useful when defining subsidiary collaborative processes that may have different membership criteria to the parent. An implementation of **Membership** is responsible for the establishment of **composition** relationships between the containing and contained **Membership** instances. The **remove_membership** operation enables the retraction of a Membership association with a parent Membership.

```
void add_membership(
    in CommunityFramework::Membership membership
);

void remove_membership(
    in CommunityFramework::Membership membership
);
```

The **get_memberships** operation enables a client to access references to the set of subsidiary **Membership** instances associated with a given **Membership**.

```
void get_memberships(
        out MembershipSequence membership_list,
```

```
        out MembershipIterator membership_iterator
) raises (
        PrivacyConflict
);
```

3.2.3.9 Membership Composition

The following (non-UML) schematic shows an example of subsidiary membership composition. A subsidiary **Membership** of the type **Encounter** (see Section 4.2.1, “Encounter,” on page 4-3) establishes a reference to the signatory **MembershipKind** as the defining model, thereby restricting the scope of the **Membership**.

While mechanisms supporting the management of composition and associations between the **Membership** and **Member** instance are implementation independent, an implementation of **Membership** or a derived type may optimize the management of **Membership** operations through selective delegation. For example, the subsidiary **Encounter** shown in this example could delegate **is_member** operations for the **MembershipKind** “signatories” to the parent **Membership**. **Memberships** and derived types could be presented as work breakdown structures, flows, or sequences of interdependent messages.

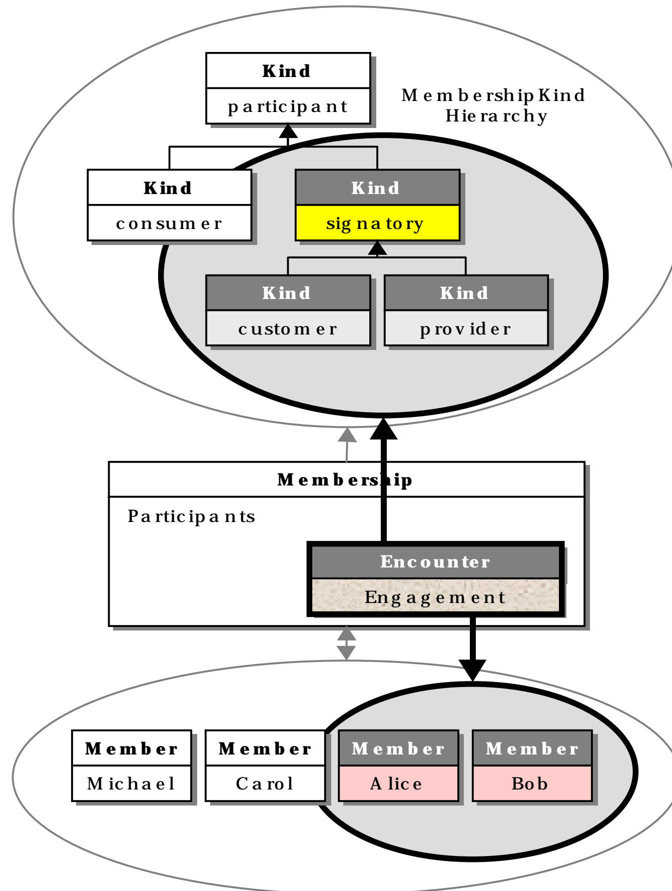

Figure 3-4 Example of a composite membership instance modifying membership criteria

3.2.4 MembershipKind

A **MembershipKind** defines the quorum, ceiling, privacy, exclusivity, permission, and association constraints associated with an instance of **Membership**.

The **quorum** level attribute indicates the required number of **Member** instances that must be added for the membership to be considered valid. The **ceiling** attribute defines the maximum number of **Member** instances that may be added, above which **Member** addition is disabled (a value of 0 indicates no limit). The **exclusive** attribute, when true indicates that no **Member** instances may reference the same **ActiveUser** identity as another **Member** instance.

The **privacy** value qualifies the extent of information disclosure enabling limitations on **Member** association disclosure as opposed to structural information (**Membership** composition). The **permission** and **association** constraints define rules concerning the association of parent and subsidiary Membership instances.

An implementation of **Membership** is responsible for the enforcement of the constraints defined within the **MembershipKind**.

3.2.4.1 IDL Specification

```
interface MembershipKind :
        SessionFramework::AbstractTemplate
        {
        readonly attribute SessionFramework::Kind kind;
        readonly attribute long quorum;
        readonly attribute long ceiling;
        readonly attribute PrivacyPolicyValue privacy;
        readonly attribute boolean exclusivity;
        enum NodeDescriptor{
            VIRTUAL_NODE,
            PHYSICAL_NODE
        };
        readonly attribute NodeDescriptor node_type;
        readonly attribute boolean exclusive;
};
```

Table 3-6 MembershipKind Attribute Table

Name	Type	Properties	Purpose
kind	Kind	read-only	String describing the Membership body.
quorum	long	read-only	An integer that defines the minimum number of Member instances that must be associated to the Membership before the Membership is considered as a valid body.
ceiling	long	read-only	An integer expressing the maximum number of Member instances that may be associated with a Membership.
privacy	PrivacyPolicyValue	read-only	Refer to Table 3-7 on page 3-16.
exclusive	boolean	read-only	The exclusive attribute, when true indicates that no Member instances may reference the same ActiveUser identity as another Member instance.
node_type	NodeDescriptor	read-only	Refer to Table 3-8 on page 3-16.

Table 3-7 PrivacyPolicyValue Enumeration Table

Value	Description
PUBLIC_DISCLOSURE	Operations may return structural and membership kind associations to non-members.
RESTRICTED_DISCLOSURE	Operations may return structural and membership kind associations to members that share a common root **MembershipKind**.
PRIVATE_DISCLOSURE	Disclosure of **MembershipKind** structure and **Member** associations is restricted to the members of the **MembershipKind** (a.k.a. private party).

Table 3-8 NodeDescriptor Enumeration TAble

Value	Description
VIRTUAL_NODE	Association by a **Membership** instance of **Members** with a **node_type** exposing **VIRTUAL_NODE** is restricted to the aggregation of the **Member** associations to subsidy **MembershipKind** stances. Invocation of **add_member** under **Membership** may raise the VirtualKind exception or return a Membership to a subsidiary kind.
PHYSICAL_NODE	A **Membership** may invoke **add_member** with the kind argument referring to a **MembershipKind** exposing this value.

3.3 Community and Derived Interfaces

3.3.1 Overview

The two interfaces **Community** and **Agency** define a framework for the management of higher level business-to-business negotiation and collaborative encounters in which the notion of organizational context and authority are intrinsic characteristics.

- **Community** extends **ActiveWorkspace** with the notion of **Membership** and the notion of a place in the context of **CosLifeCycle** “here” and “there.”
- An **Agency** extends the notion of **Community** through the introduction of **LegalEntity** and thereby authority through **Jurisdiction**.

3.3.1.1 Object Model

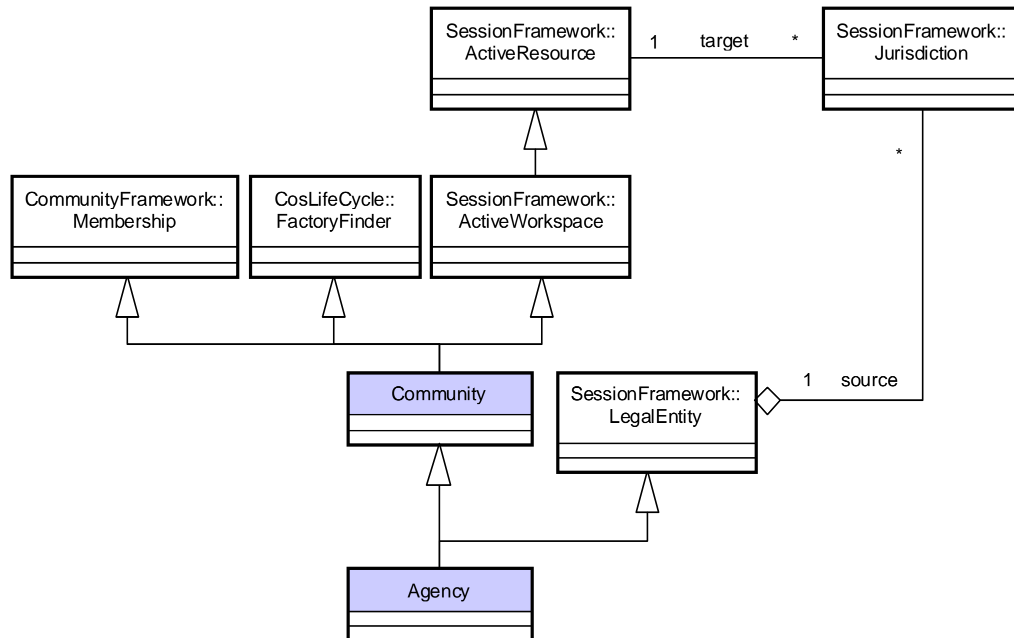

Figure 3-5 Community, Agency, and Jurisdiction Object Model

3.3.2 Community

A **Community** extends the notion of workspace through the introduction of **CosLifeCycle::FactoryFinder** and **Membership**. As a **FactoryFinder**, a **Community** enables client applications controlled access to resources that may be required during the course of a collaborative encounter (relative to a collaborative context) and the ability to publish resources into a **Community** (where a **Community** constitutes the **there** argument to **LifeCycleObject copy** or **move** operation). As a **Membership**, a **Community** is associated to constraints concerning quorum, ceiling, privacy, and associative constraints.

3.3.2.1 IDL Specification

```
interface Community :
        SessionFramework::ActiveWorkspace,
        CosLifeCycle::FactoryFinder,
        Membership
{
};
```

3.3.3 Agency

Agency is a specialization of **Community** and **LegalEntity** that introduces the notion of organized community such as a company. As a **LegalEntity**, an **Agency** may be associated under a **Jurisdiction** relationship over a set of resources. Client applications may navigate the **Jurisdiction** relationship in order to qualify the context of collaboration and authority of respective participants.

3.3.3.1 Object Model

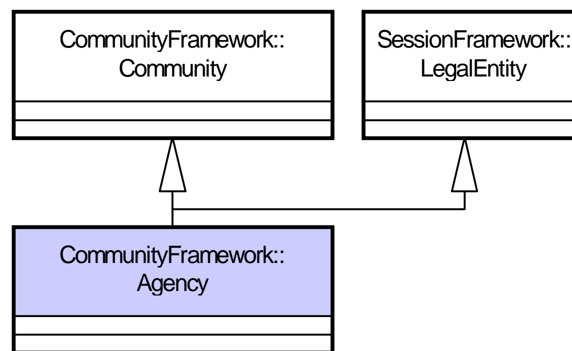

Figure 3-6 Agency Object Model.

3.3.3.2 IDL Specification

```
interface Agency :
        Community,
        SessionFramework::LegalEntity
{
};
```

3.3.4 CommunityFramework IDL

```
// File: CommunityFramework.idl

#ifndef _COMMUNITY_FRAMEWORK_IDL_
#define _COMMUNITY_FRAMEWORK_IDL_
#pragma prefix "omg.org"

#include <SessionFramework.idl>

module CommunityFramework{

        // forward declarations
```

```
interface MembershipKind;
interface Membership;
interface Member;

interface Community;
interface Agency;

// typedefs

typedef sequence <Member> MemberSequence;
typedef sequence <Membership> MembershipSequence;
typedef sequence <MembershipKind> MembershipKindSequence;

// iterators

interface MemberIterator : CosCollection::Iterator{};
interface MembershipIterator : CosCollection::Iterator{};
interface MembershipKindIterator : CosCollection::Iterator{};

// Membership


enum PrivacyPolicyValue {
        PUBLIC_DISCLOSURE,
        RESTRICTED_DISCLOSURE,
        PRIVATE_DISCLOSURE
};

exception PrivacyConflict{
        PrivacyPolicyValue reason;
};

interface MembershipKind :
        SessionFramework::AbstractTemplate
        {
        readonly attribute SessionFramework::Kind kind;
        readonly attribute long quorum;
        readonly attribute long ceiling;
        readonly attribute PrivacyPolicyValue privacy;
        readonly attribute boolean exclusivity;
        enum NodeDescriptor{
                VIRTUAL_NODE,
                PHYSICAL_NODE
    };
    readonly attribute NodeDescriptor node_type;
    readonly attribute boolean exclusive;
};

interface Membership :
    SessionFramework::ActiveResource {
```

```
readonly attribute MembershipKind model;
enum RecruitmentStatus{
        OPEN_MEMBERSHIP,
        SUSPENDED_MEMBERSHIP,
        CLOSED_MEMBERSHIP
};
readonly attribute RecruitmentStatus recruitment_status;
exception RecruitmentConflict{
        RecruitmentStatus reason;
};
enum QuorumStatus {
        QUORUM_REACHED,
        QUORUM_PENDING,
        QUORUM_UNREACHABLE
};
readonly attribute QuorumStatus quorum_status;
readonly attribute long count;
readonly attribute long active_count;
exception AttemptedCeilingViolation{ };
exception AttemptedExclusivityViolation{ };
exception VirtualKind{ };
exception UnknownKind{ };
exception MembershipRejected{
        Membership source;
        string reason;
};
Member add_member(
        in SessionFramework::ActiveUser user,
        in CommunityFramework::MembershipKind kind
) raises (
        AttemptedCeilingViolation,
        AttemptedExclusivityViolation,
        RecruitmentConflict,
        MembershipRejected,
        UnknownKind,
        VirtualKind
);
boolean is_member(
        in CommunityFramework::Member member,
        in CommunityFramework::MembershipKind kind
) raises (
        PrivacyConflict
);
void get_members(
        in MembershipKind kind,
        out MemberSequence member_list,
        out MemberIterator member_iterator
) raises (
        PrivacyConflict
);
void remove_member(
```

```
                in CommunityFramework::Member member
        );
        void add_membership(
                in CommunityFramework::Membership membership
        );
        void get_memberships(
                out MembershipSequence membership_list,
                out MembershipIterator membership_iterator
        ) raises (
                PrivacyConflict
        );
        void remove_membership(
                in CommunityFramework::Membership membership
    );
};

interface Member :
        SessionFramework::Delegation,
        SessionFramework::ActiveUser
        {
        attribute string label;
        readonly attribute CommunityFramework::Membership
                membership;
    void get_kinds(
            out MembershipKindSequence kind_list,
            out MembershipKindIterator kind_iterator
    ) raises (
            PrivacyConflict
    );
};

interface Community :
        SessionFramework::ActiveWorkspace,
        CosLifeCycle::FactoryFinder,
        Membership {
};

interface Agency :
        Community,
        SessionFramework::LegalEntity {
};

}; // end CommunityFramework Module

#endif // _COMMUNITY_FRAMEWORK_IDL_
```

Collaboration Framework 4

Contents

This chapter contains the following topics.

Topic	**Page**
"Overview"	4-1
"Encounter and Associated Interfaces"	4-3
"Collaboration Interfaces"	4-8
"Negotiation and Promissory Models"	4-30

4.1 Overview

The **CollaborationFramework** module is composed of three distinct groups of interfaces:

1. Base interfaces defining an **Encounter** process and an associated **EncounterTemplate.**
2. Interfaces supporting **Collaboration**, **Engagement**, and **Voting** encounters.
   - **Collaboration**: an **Encounter** that enables execution of negotiation and promissory models such as those defined under the Session Framework section of this specification.
   - **Voting**: an **Encounter** used to aggregate votes in the determination of a success or fail condition.
   - **Engagement**: an **Encounter** used to establish a contractual agreement across a set of participant Members.
3. Interfaces managed by the **CollaborationTemplate** type.

The **CollaborationFramework** builds above interfaces defined under the **CommunityFramework** and **SessionFramework**.

Table 4-1 Base Interfaces of the CollaberationFramework Module

Interface	Description
Encounter	A specialization of **ActiveTask** and **Membership** that has an association to an **EncounterTemplate** that defines the encounter constraints, and an associated **subject**.
EncounterTemplate	A specialization of **AbstractTemplate** that references a **MembershipKind** applicable to an **Encounter** of the type described by **EncounterTemplate**.

Table 4-2 Interfaces Derived from Encounter and EncounterTemplate

Interface	Description
Collaboration	A type of **Encounter** bound to a **CollaborationTemplate** that mediates access to a **subject**. **Collaboration** exposes the state of a collaborative process and brings together the operations that may be applied by collaborating users relative to a process **template**. An **apply** operation enables the invocation of simple and compound transitions that under the mediated control of the Collaboration enable parties to reach terminal **success** or **failure** states. The **active-state** of **Collaboration** is a reference to a sequence of **State** instances held within the associated **template**. Users are associated to a **Collaboration** through a **Member** role.
Voting	A type of **Encounter** launched by a compound transition supporting vote-based determination of primary or alternate state selection. **Voting** is an interface that provides mechanisms through which users in a collaborative process can register **YES**, **NO**, or **ABSTAIN** votes. **VoteTemplate** exposes policies concerning quorum and structured numerator/denominator pair that defines the required ceiling for calculation of a successful vote.
Engagement	A type of **Encounter** defined by an associated **EngagementTemplate** that enables the association of proof of engagement to an agreement. Features associated to **EngagementTemplate** define the security criteria to be applied during the engagement process. **EngagementManifest** is a type supporting the registration of proof as defined by the **EngagementTemplate**.

Table 4-3 CollaborationTemplate Dependent Interfaces

Interface	Description
State	A type that exposes a **label**, characteristics that qualify the state as **internal**, terminal **success**, or terminal **failure** exposes a set of **sub-states** and **parent** state.

Table 4-3 CollaborationTemplate Dependent Interfaces

Trigger	A type that exposes a **keyword**, accesses, and timeout constraints. Triggers are used as a super-type for the **Command** and **Transition** types. Operational qualifiers include a usage **mode** and references to a **MembershipKind** that is authorized to invoke a **Trigger**. Usage **mode** enables the declaration of constraints over activation relative to the collaborative context.
Command	A specialization of **Trigger** that enables the declaration of an event that may be invoked under **Collaboration**.
Transition	A Transition extends **Trigger** to include a source and destination state. A transition may only be invoked when the **active-state** of collaboration is the **source** state in the **Transition** declaration. Following a successful activation of a transition, the destination state and all parents of the destination state are considered active by the controlling **Collaboration**.
CompoundTransition	A specialization of **Transition** that introduces an alternative destination **State** and **template** describing the criteria for **Encounter** creation. **CompoundTransition** provides a powerful mechanism to express recursive collaborative encounters such as amendments under multilateral negotiation.

4.2 Encounter and Associated Interfaces

4.2.1 Encounter

An **Encounter** is an abstract type that exposes the run-time state of collaborative process involving a collection of participating members. **Encounter** is the super-type of **Collaboration**, **Voting**, and **Engagement**. An **Encounter** is created by a generic factory, using the features exposed by an associated **EncounterTemplate** as the factory **key** and **criteria**. An **EncounterTemplate** defines the policy applicable to the Encounter and may reference a required **MembershipKind**. **Encounter** is derived from **Membership** and as such represents a collection of users, bound together as members of the **Encounter**.

4.2.1.1 Object Model

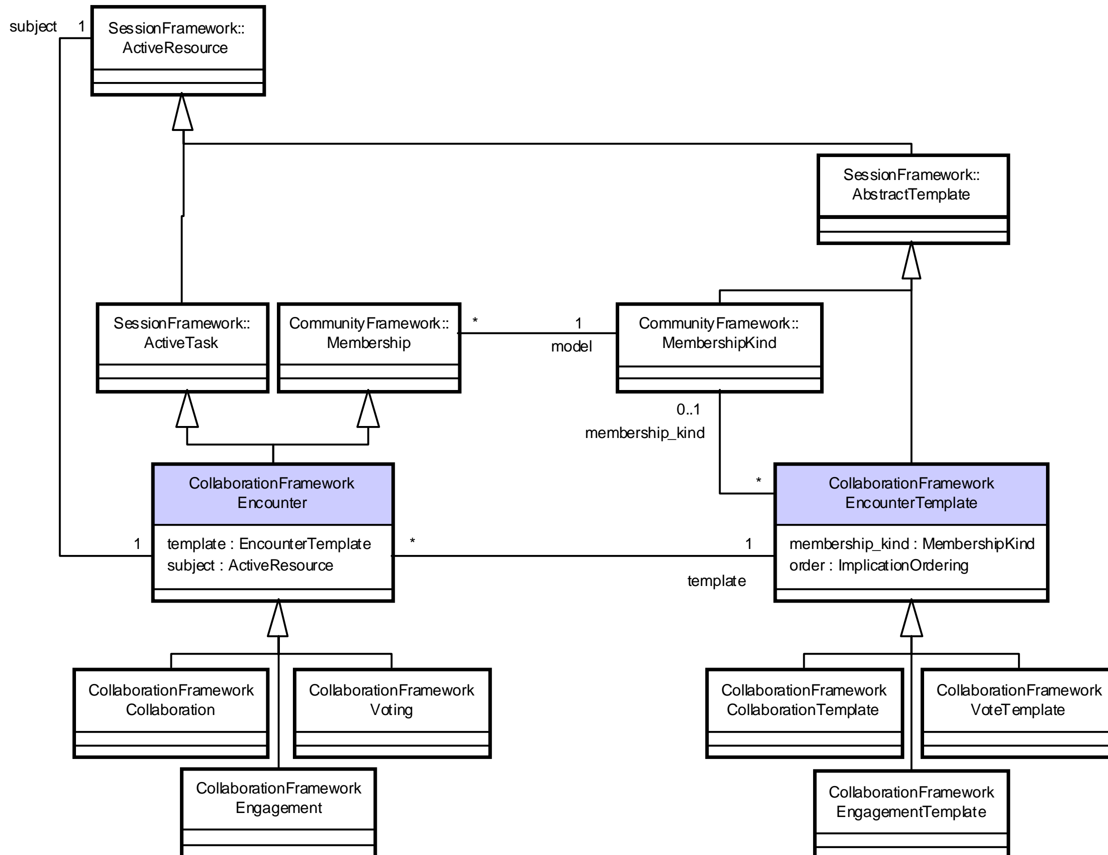

Figure 4-1 Encounter and EncounterTemplate Object Model

4.2.1.2 IDL Specification

```
interface Encounter :
    CommunityFramework::Membership,
    SessionFramework::ActiveTask
    {
    readonly attribute EncounterTemplate template;
    readonly attribute SessionFramework::ActiveResource subject;
};
```

Table 4-4 Encounter Attribute Table

Name	Type	Properties	Purpose
subject	ActiveResource	read-only	A reference to an **ActiveResource** that constitutes the **subject** of the **Encounter**.
template	EncounterTemplate	read-only	A reference to an **EncounterTemplate** that exposes the **membership_kind** to be used by the **Encounter**.

Table 4-5 Encounter Structured Event Table

Event	Description
Result	Notification of success or failure of execution of an encounter. <u>Supplementary properties</u>: value boolean True indicates that the task concluded with a successful result. False indicates that the task failed. Determination of success or failure is a function of a specialization of the Encounter type.

4.2.1.3 Initialization

The **template** attribute exposed by **Encounter** refers to an **EncounterTemplate** that qualifies the kind of **Encounter** and applicable constraints. On instantiation of an **Encounter**, an implementation is responsible for the association of the **Encounter** to a **MembershipKind** using the attribute **model** inherited from **Membership**. The value attributed to **model** at runtime is the value of **membership_kind** exposed by the associated **EncounterTemplate**.

4.2.1.4 Implication Semantics

An **Encounter** associated with **EncounterTemplate** that exposes an **Implication** association is, on completion, following the raising of a success or failure **result** event, required to establish instances of Encounter as referenced by appropriate **Success** or **[Reviewer, changed of to or, please verify] Failure** implications. Encounter success will raise **Success** implications whereas **Encounter** failure will raise **Failure** implications. **Implications** are executed as a set of sub-processes to the current **Encounter** during which time the hosting **Encounter** enters a suspended state.

4.2.2 Encounter Template

An **EncounterTemplate** is an abstract type that exposes **membership_kind**. This is used by an instance of **Encounter** during initialization to establish the **MembershipKind** to be bound to the **model** attribute inherited from **Membership**.

4.2.2.1 IDL Specification

```
interface EncounterTemplate :
    SessionFramework::AbstractTemplate
    {
    readonly attribute CommunityFramework::MembershipKind
            membership_kind;
    enum ImplicationOrdering {
        SEQUENTIAL,
        PARALLEL
    };
    readonly attribute ImplicationOrdering order;
};
```

Table 4-6 EncounterTemplate Attribute Table

Name	Type	Properties	Purpose
membership_kind	MembershipKind	read-only	A reference to a **MembershipKind** that defines the value to be assigned to the **model** attribute of an **Encounter**. Used to qualify the membership **kind** required to participate to an **Encounter**.
order	ImplicationOrdering	read-only	Refer to Table 4-7 on page 4-6.

Table 4-7 ImplicationOrdering Enumeration Table

Value	Description
SEQUENTIAL	An instance of Encounter is responsible for the creation and execution of subsidiary Encounter instances in accordance with the Implication references in **sequential** order.
PARALLEL	An instance of Encounter is responsible for the creation and execution of subsidiary Encounter instances in accordance with the Implication references in **parallel**.

4.2.3 Implication

An **Implication** is an abstract specialization of **Linkage** (see Section 2.2.2, “Linkage,” on page 2-8). **Implication** associates a **source** instance of **EncounterTemplate** with a **target** instance of **EncounterTemplate**. Two concrete

types of **Implication** include **Success** and **Failure** that may be used by client applications such as **Encounter** to manage the instantiation of sub-process that correspond to the consequences of a successful or unsuccessful process.

4.2.3.1 *Object Model*

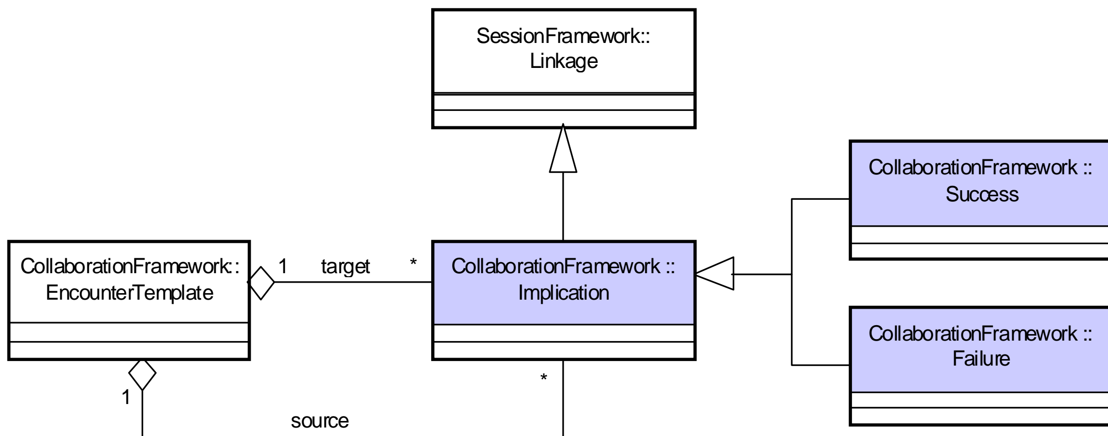

Figure 4-2 Implication and the derived types Success and Failure

4.2.3.2 *IDL Specification*

```
interface Implication :
SessionFramework::Linkage
{
};

interface Success :
    Implication
{
};

interface Failure :
Implication{
};
```

4.3 Collaboration Interfaces

4.3.1 Collaboration

Collaboration is a concrete specialization of **Encounter** whose semantics are defined by the type **CollaborationTemplate**. The model is expressed as a state-set composed of sub-states, transitions, and interaction constraints.

The **Collaboration** type enables users to invoke transition operations that lead from initial to terminal states. Customizable process models allow the introduction of semantics dealing with collaborative processes typified by the bilateral negotiation, multilateral negotiation, and promissory engagement models discussed under the Session Framework section of this specification.

A client joins an instance of **Collaboration** by establishing a **Member** role and associating the role to **Collaboration** using the **add_member** operating inherited from **Membership**. Clients interact with the collaboration through the operations **apply** and **invoke**. The **apply** operation tasks three arguments: **transition**, a **semantic** qualifier, and a reference to a **task** that may be bound as **producer** of the **subject** of the collaboration, or alternatively, may invoke a replacement of the **subject** of the **Collaboration** (depending on the **semantic** qualifier).

The task argument is used to establish a **Member** as the active editor of the **subject** of the collaboration. In the case of subject modification, the client task is associated as **producer** by the **Collaboration**. The producer relation between client task and subject is maintained until (a) the client relinquishes the **producer**, or (b) the **Collaboration** retracts the **producer** relationship from the client.

4.3.1.1 Object Model

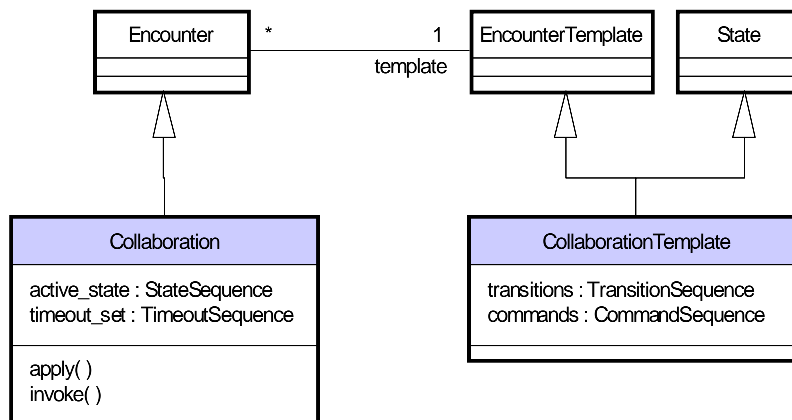

Figure 4-3 Collaboration Object Model.

4.3.1.2 IDL Specification

```
interface Collaboration :
        Encounter
        {
        readonly attribute StateSequence active_state;
        struct TimeoutSequence{
            CollaborationFramework::Trigger trigger;
            TimeBase::UtcT timestamp;
        };
        readonly attribute TimeoutSequence timeout_list;
        exception InvalidTrigger{
            CollaborationFramework::Trigger trigger;
        };
        exception ApplyFailure{
            CollaborationFramework::Trigger trigger;
            SessionFramework::ActiveTask task;
        };
        enum ApplySemantics{
            REPLACEMENT,
            MODIFICATION,
        };
        exception SemanticConflict{ };
        void apply(
            in CollaborationFramework::Transition transition,
            in ApplySemantics semantic,
            in SessionFramework::ActiveResource resource
        ) raises (
            InvalidTrigger,
            SemanticConflict,
            ApplyFailure
        );
        void invoke(
            in SessionFramework::Command command,
            in SessionFramework::ActiveResource argument,
            in string reason
        ) raises (
            InvalidTrigger
        );
};
```

Table 4-8 Collaboration Attribute Table

Name	Type	Properties	Purpose
active_state	StateSequence	read-only	An ordered sequence of instances of **State**. The sequence order is from most general to most specific. States exposed in the sequence are derived from **apply** operations invoked on **Collaboration** associated to a **CollaborationTemplate**.
timeout_list	TimeoutSequence	read-only	**Triggers** exposed by a **CollaborationTemplate** may declare timeout behavior. **Triggers** with timeout behavior are considered active if their source state is an active state. As several states may be active at any time and for each active state, there may exist several timeout transitions. The **timeout_list** attribute exposes all active timeout transition declarations. An implementation of **Collaboration** is responsible for the applying timeout transition and managing the **timeout_list**.

Table 4-9 Collaboration Structured Event Table

Event	Description
Inform	Command event raised as a result of an invoke operation. Supplementary properties: keyword keyword Keyword associated with the Command object referenced by the invoke operation.

4.3.1.3 Relationship to Collaboration Template

CollaborationTemplate is a process model that defines the semantic conditions of **Collaboration**. A **CollaborationTemplate** is a specialization of a **State** and **EncounterTemplate** that exposes a set of transitions and command events that may be applied/invoked by an instance of **Collaboration**. As a **State**, a **CollaborationTemplate** exposes a sub-state hierarchy. Transitions exposed by **CollaborationTemplate** are declarations of source and destination states and inherit activation constraints from the super-type **Trigger**. **Trigger** defines activation constraints based on collaborative context and user's membership, and in the case of **Transition**, the implications of the transition relative to the **subject** of collaboration.

4.3.1.4 Subsidiary Collaboration Processes

CompoundTransition declarations reference **EncounterTemplate** instances that may be by the **Collaboration** to create subsidiary **Encounter** processes to an instance of **Collaboration**. An **EncounterTemplate** exposes references to a type supporting the execution of the template. For example, **CollaborationTemplate** is executed by **Collaboration**, **VoteTemplate** is executed under **Voting**, and an **EngagmentTemplate** is executed under **Engagement**. These associations are exposed by a **factory_key** on **AbstractTemplate**.

Collaboration, **Voting**, and **Engagement** are examples of specialized **Encounter** types that under the management of a **Collaboration** resolve in success or failure conditions that determine the behavior of compound transitions managed by the host **Collaboration**.

4.3.1.5 Active State

The active-state of **Collaboration** is a function of the **apply** operations invoked against a **Collaboration** within the scope of an associated **CollaborationTemplate**. On invocation of the **apply** operation a **Transition** is passed in as an argument. On successful completion of the transition, the transition **target** state and all **parent** states of the **target** define the active state of the **Collaboration**. For example, if state C is referenced as the target, and C references the parent state B and B references a parent state A, the active state sequence will be the order sequence of states A, B, and C.

```
readonly attribute StateSequence active_state;
```

The **active_state** of **Collaboration** is used to determine active **Trigger** instances. **Trigger** instances are considered active when a **Trigger** source state is itself active. An implementation of Collaboration maintains the **active_state** value.

4.3.1.6 Timeout behavior

Timeout behavior is defined by instances of **Trigger** that has a non-null timeout value. An implementation of Collaboration is required to maintain the value of the **timeout_list** attribute such that it contains only active timeout triggers and the associated timestamp. The value of timestamp corresponds to the time when the associated **State** became active. **Trigger** and **timestamp** pairs are captured under the structure **TimeoutStructure**. A sequence of active **TimeoutStructure** values is exposed by the **timeout_list** attribute. Changes to the **timeout_list** must be signalled by an **update** event (see Section 2.2.1.5, “Structured Events,” on page 2-7).

```
struct TimeoutSequence{
        CollaborationFramework::Trigger trigger;
        TimeBase::UtcT timestamp;
};

readonly attribute TimeoutSequence timeout_list;
```

4.3.1.7 Initialization of a Collaboration

Collaboration is considered as non-initialized if the **active_state** returns an empty sequence. Initialization is achieved by invoking the **apply** and passing a **Transition** that exposes a **TRUE** value under the **initialize** attribute.

4.3.1.8 Applying State Transitions

Application of a state transition is the mechanism used to change the context of the collaboration and potentially replace or modify the **subject** of **Collaboration**. For example, a **CollaborationTemplate** exposing the sub-states SCHEDULE and DELIVERED could associate the two states through a transition named “deliver.” The transition “delivered” could be attributed with the following characteristics:

- access constraints based on membership kind
- constraints that impose restrictions based on collaborative context
- declaration of the usage of the resource argument by a Collaboration during a transition

Collaborative context and membership kind constraints collectively guard a Trigger invocation. Enforcement of these constraints is the responsibility of an implementation of the **apply** operation. The behavior of **apply** under a simple transition is determined by the **Transition** referenced under the **transition** argument, and in the case of **PROCESS** based **Transition**, a qualifying semantic argument and task argument.

Two constraints exist within a Transition:

1. A collaborative context guard that restricts the invoking principal to the **INITIATOR**, a **RESPONDENT**, or **PARTICIPANT** (where **PARTICIPANT** is the superset of **INITIATOR** and **RESPONDENT**).

2. Once the collaborative context and any Membership restrictions are satisfied, an **apply** implementation can evaluate the kind of transition being invoked.

Four kinds of transitional behavior are exposed by the **Transition** instance under the **control** attribute. These behaviors are **FAIL**, **RESET**, **TRANSITIONAL**, and **PROCESS** transitions. A **FAIL** transition is a null transition and terminates without change to the process. A **RESET** transition is equivalent to **FAIL**; however, the state referenced by the **source** is re-entered and associated timeouts are reset. **TRANSITIONAL** results in the establishment of a new active state sequence based on the transition **target** and **parent** states. In the case of a **PROCESS** transition, the **semantic** and **resource** arguments are taken into consideration.

```
enum ApplySemantics{
        REPLACEMENT,
        MODIFICATION,
};

void apply(
        in CollaborationFramework::Transition transition,
        in ApplySemantics semantic,
```

```
        in SessionFramework::ActiveResource resource
) raises (
        InvalidTrigger,
        ActiveTaskTypeConflict,
        ApplyFailure
);
```

Table 4-10 ApplySemantics Enumeration Table

Value	Description
REPLACEMENT	The **resource** argument under the **apply** operation constitutes a replacement of the **subject** of the **Collaboration** (conditional to transition constraints).
MODIFICATION	The **resource** argument under the **apply** operation constitutes a **ReactiveTask** that is to be bound as **producer** of the **subject** of the **Collaboration** in order to invoke changes to the **subject** (conditional to transition constraints).

If the semantic argument is the enumerated value **REPLACEMENT**, then the **resource** argument constitutes an **ActiveResource** to be established as a new **subject** value. If the semantic argument is **MODIFICATION**, then the **resource** argument is an **ActiveTask** that will be bound as **producer** of the **subject** of the **Collaboration**. The subsidiary modification task will execute, complete, and return the **produces** association to the host. The host will then complete the transition by setting the **active_state** value.

The sequence of rules processing concerning the management of apply in the context of membership, collaborative context, transition controls that may be present at more than one level (as is the case of a **PROCESS** based **CompoundTransition**) are detailed under the following three rules.

RULE 1: Evaluate guard conditions

- verify the collaborative **context** rights
- verify membership **kind** rights

RULE 2: Establish path

For a simple **Transition**, the **control** and **target** State are established from values exposed by the **transition** argument.

For a **CompoundTransition**:

- Create the **Encounter** sub-process and bind the host **subject** as sub-process **subject** if needed.
- Select initialization.
- Invoke **apply** using selected initializing **transition**, and the semantic and resource arguments from the host **apply** operation.
- Wait for **Encounter** sub-process completion and evaluate success or fail **result** status.

- Select the **control** value and **target** state based on **result** status.

RULE 3: Execute in accordance with the transition control criteria

- Under the **FAIL** criteria a transition is complete, no change to active state or subject is effected.
- Under the **RESET** criteria the implementation is required to reset any active **TIMEOUT** transitions. No change to active state or subject is effected. Transition is complete.
- Under the **TRANSITIONAL** criteria an implementation sets the root active state to the **target** state. No subject change is effected. Transition is complete.
- Under the **PROCESS** criteria the following conditions apply:
  - If the transition is a **CompoundTransition** under either the **MODIFICATION** or **REPLACEMENTS** semantic, then assign the **subject** of the subsidiary **Encounter** to be the **subject** of the host **Encounter**.
  - If the transition is a simple **Transition**, then
    - Under **REPLACEMENT** semantics, the resource argument is assigned as a new host subject.
    - Under **MODIFICATION** semantics, the resource argument is a task that will be associated as producer of the host subject, executes (causing changes to the host subject) and completes, following which the host will reassign the produces relationship to itself.

4.3.1.9 Apply Exceptions

If, during the invocation of the apply operation where the **semantic** argument is **PROCESS**, and the type of object passed under the **resource** argument is not an **ActiveTask** or type derived from **ActiveTask**, an ActiveTaskTypeConflict exception will be raised.

```
exception ActiveTaskTypeConflict{
        ActiveResource resource;
};
```

The InvalidTrigger exception may be raised if the **Trigger** passed in under the **trigger** argument is not in the **active_path** of the **Collaboration**.

```
exception InvalidTrigger{
        CollaborationFramework::Trigger trigger;
};
```

In the case of the failure of the execution of a task executing in the context of **MODIFICATION** semantics, the ApplyFailure exception may be raised.

```
exception ApplyFailure{
        CollaborationFramework::Trigger trigger;
        SessionFramework::ActiveTask task;
};
```

4.3.1.10 Invoking Command Events

Command instances describe events that may be raised by the **invoke** operation on **Collaboration**. Access constraints enforced by an implementation of **invoke** are defined by the features exposed on the inherited **Trigger** interface (see Section 4.3.3, “Trigger,” on page 4-18). The event raised by invoking a **Command** object is exposed as an **inform** event type with **keyword**, **reason,** and **argument** properties corresponding to the **Command keyword** and invoke arguments.

```
void invoke(
        in CosObjectIdentity::ObjectIdentifier id,
        in SessionFramework::ActiveResource argument,
        in string reason
) raises (
        InvalidTriggerIdentity
);
```

4.3.2 CollaborationTemplate

CollaborationTemplate is a specialization of a **State** and **EncounterTemplate** that exposes a set of transition declarations that may be applied to an instance of **Collaboration**. As a **State**, a **CollaborationTemplate** exposes a sub-state hierarchy that enables the activation of command events and transition. Transitions exposed by **CollaborationTemplate** are declarations of source and destination states that may be used as arguments under the **Collaboration** interface **apply** operation.

Both **Command** and **Transitions** references exposed by **CollaborationTemplate** inherit activation constraints from the super-type **Trigger**. **Trigger** defines activation constraints based on collaborative context and user’s membership, and in the case of **Transition**, the implications of the transition relative to the **subject** of collaboration.

4.3.2.1 Object Model

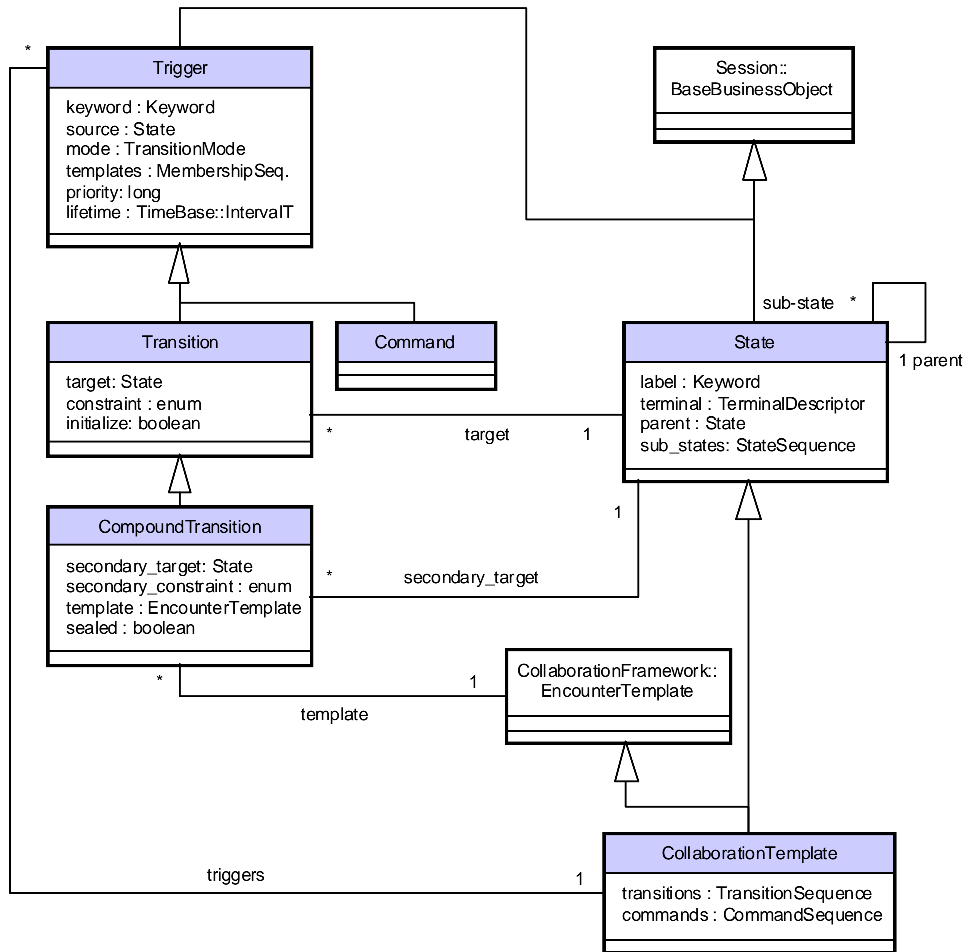

Figure 4-4 State, CollaborationTemplate, Trigger, Command & Transition Interfaces

4.3.2.2 IDL Specification

```
interface CollaborationTemplate :
        EncounterTemplate,
        State
        {
        readonly attribute TransitionSequence transitions;
        readonly attribute CommandSequence commands;
};
```

Table 4-11 CollaborationTemplate Attribute Table

Name	Type	Properties	Purpose
transitions	TransitionSequence	read-only	A sequence of Transition instances. Transition instances exposed under this attribute together with State models enable client application to construct process descriptions.
commands	CommandSequence	read-only	A sequence of Command instances that are managed by the Collaboration template.

4.3.2.3 State

A **State** is a type contained within a **CollaborationTemplate**. A **State** exposes a unique **Keyword** within the scope of a **CollaborationTemplate**, maintains references to **sub_states**, and contains a reference to a parent **State**. The attribute **terminal** characterizes a state as a terminal, indicating the completion of a **Collaboration** process in a **SUCCESS** or **FAILURE** condition.

The primary function of a **State** is to support the expression of a collaborative process model. The **active_state** of a **Collaboration** is a sequence of **State** instances that include the most general parent **State**, through to the most specific **State** (see Section 2.2, "ActiveResource and Associative Interfaces," on page 2-5). The **parent** and **sub-state** attributes of a **State** allow client applications to navigate a **State** hierarchy.

4.3.2.4 IDL Specification

```
typedef string Keyword;

interface State :
        Session::BaseBusinessObject
{
        readonly attribute Keyword label;
        enum TerminalDescriptor{
                INTERNAL,
                SUCCESS,
                FAILURE
        };
        readonly attribute TerminalDescriptor terminal;
        readonly attribute State parent;
        readonly attribute StateSequence sub_states;
};
```

Table 4-12 State Attribute Table

Name	Type	Properties	Purpose
label	Keyword	read-only	A text string describing the **State** (such as offered, proposed, requested).
terminal	TerminalDescriptor	read-only	An enumeration that qualifies that **State** as terminal or non-terminal (see Table 4-13 on page 4-18).
parent	State	read-only	A reference to the **State** to which this **State** is subsidiary. The **parent State** must expose a reference to this **State** in its **sub_state** attribute.
sub_states	StateSequence	read-only	A sequence of **State** instances that are subsidiary to this **State**.

Table 4-13 TerminalDescriptor Enumeration Table

Name	Type
INTERNAL	Establishes the state as a non-terminal state.
SUCCESS	Establishes the state as a terminal state in which the process is classified as a success.
FAILURE	Establishes the state as a terminal state in which the process is classified as a failure.

4.3.2.5 State Composition

States are composed by association of sub-states to a parent state. A parent state exposes sub-states through the **sub_states** attribute that returns a sequence of sub-state references. A **State** has exactly one **parent** State (possibly itself in the case of a top level State). Sub-states enable navigation to their parent state through the **parent** attribute.

```
readonly attribute State parent;
readonly attribute StateSequence sub_states;
```

4.3.3 Trigger

A **Trigger** is a base type for **Command** and **Transition** types. A **Trigger** may be invoked under explicit activation by a user through the **apply** or **invoke** operation under the **Collaboration** interface, or by an implementation of **Collaboration** through association of a **lifetime** value. Timeout based execution is enabled if the **lifetime** attribute contains a non-null value. A **Trigger** exposes a **source State**. When the **source State** is referenced in the **active_state** chain of a collaboration, the **Trigger** is considered active.

4.3.3.1 IDL Specification

```
interface Trigger :
        Session::BaseBusinessObject{
        readonly attribute CollaborationFramework::Keyword keyword;
        enum TriggerMode{
                INITIATOR,
                RESPONDENT,
                PARTICIPANT,  TIMEOUT
        };
        readonly attribute State source;
        readonly attribute TriggerMode mode;
        readonly attribute CommunityFramework::MembershipKind
                                                constraint;
        readonly attribute long priority;
        readonly attribute TimeBase::IntervalT lifetime;
};
```

Table 4-14 Trigger Attribute Table

Name	Type	Properties	Purpose
keyword	Keyword	read-only	A string used to label the transition.
source	State	read-only	The **State** that must be exposed under a **Collaboration active_state** for the **Trigger** to be considered usable by a **Collaboration**. The **source** constitutes the **State** to which the **Trigger** is assigned.
mode	TriggerMode	read-only	Refer to Table 4-15 on page 4-20.
constraint	MembershipKind	read-only	Defines the required membership **kind** that a user must hold in order to invoke a **Trigger**.
priority	long	read-only	A value indicating the **priority** of a **Trigger**. The default value of 0 indicates normal priority. Higher values indicate increasing priority.
lifetime	TimeBase::IntervalT	read-only	See Section 4.3.3.4, “Trigger Lifetime and Activation Semantics,” on page 4-20.

4.3.3.2 Collaborative Context and Execution Modes

A **mode** attribute qualifies the contextual role of a participant authorized to invoke a transition. The mode signifies either **TIMEOUT**, or the collaborative context enumeration values **INITIATOR**, **RESPONDENT,** and **PARTICIPANT**. A **TIMEOUT** transition is a declaration of a state transition that is invoked by the implementation. Of

the collaborative context modes, **INITIATOR** limits access to the user that established the currently active state. **RESPONDENT** refers to any participant other than the initiator. **PARTICIPANT** refers to either **INITIATOR** or **RESPONDENT**.

```
enum TriggerMode{
        INITIATOR,
        RESPONDENT,
        PARTICIPANT,
        TIMEOUT
};
```

readonly attribute TriggerMode mode;

Table 4-15 TriggerMode Enumeration Table

Value	Description
INITIATOR	**INITIATOR** mode restricts the activator of a transition to the same principal identity that invoked the last transition.
RESPONDENT	**RESPONDENT** is any **Member** within a Membership other than the principal as defined by **INITIATOR**.
PARTICIPANT	An **INITIATOR** or **RESPONDENT**.
TIMEOUT	Invocation of the **Trigger** is controlled by the implementation in accordance with the **lifetime** exposed by the **Trigger**.

4.3.3.3 Access control based on Membership

Access to a **Trigger** exposing the modes **PARTICIPANT**, **INITIATOR**, or **RESPONDENT** may be qualified further by the addition of **Membership** references under the **constraint** attribute. An invoking user must be a **Member** of the membership **kind** referenced by the constraint.

readonly attribute CommunityFramework::MembershipKind constraint;

4.3.3.4 Trigger Lifetime and Activation Semantics

A **Trigger** exposing a non-null lifetime value will be invoked automatically by an implementation of Collaboration on expiry. Timeout of a trigger is determined by the time of the last reactivation of the source state plus the time period identified under the lifetime attribute. An implementation of Collaboration exposes timeout triggers and deadlines under the **timeout_list** attribute.

readonly attribute TimeBase::IntervalT lifetime;

4.3.4 Command

A **Command** type is a specialization of **Trigger** that enables the declaration of an event that may be invoked under a **Collaboration** using the **invoke** operation.

4.3.4.1 IDL Specification

```
interface Command :
        CollaborationFramework::Trigger
{
};
```

4.3.5 Transition

A **Transition** is a type of **Trigger** that exposes a **target State** and constraints concerning the effect of a transition relative to a **subject** of **Collaboration**. Transitions that reference a source state that is active (see Section 4.3.1.5, “Active State,” on page 4-11) are themselves considered active in that they may be invoked subject to the access constraints imposed by the features inherited from **Trigger**. A **Transition** is applied to a **Collaboration** through the **apply** operation. Specification of the relationship between **ControlDescriptor constraint** and the subject of **Collaboration** is detailed under Section 4.3.1, “Collaboration,” on page 4-8.

4.3.5.1 IDL Specification

```
interface Transition :
        Trigger
        {
        enum ControlDescriptor{
                PROCESS,
                TRANSITIONAL,
                RESET,
                FAIL
        };
        readonly attribute State target;
        readonly attribute ControlDescriptor control;
        readonly attribute boolean initialize;
};
```

Table 4-16 Transition Attribute Table

Name	Type	Properties	Purpose
target	State	read-only	The **State** which will be made the root active state on successful completion of a transition.

Table 4-16 Transition Attribute Table

control	ControlDescriptor	read-only	**ControlDescripter** exposing one of the enumerated values **PROCESS**, **TRANSITIONAL**, or **FAIL** used by the apply operation on Collaboration (see Section 4.3.5.2, "Subject Modification Constraints," on page 4-22).
initialize	boolean	read-only	A value of true indicates that the transition may be invoked as an initialization, bypassing any source State constraint.

4.3.5.2 Subject Modification Constraints

A **Transition** exposes the enumerated values of **PROCESS**, **TRANSITIONAL**, and **FAIL**. These values are used by an implementation of the **apply** operation to determine behavior concerning the launching of sub-process and the potential commit or rollback of changes on completion of a transition (see Section 4.3.1, "Collaboration," on page 4-8).

```
enum ControlDescriptor{
        PROCESS,
        TRANSITIONAL,
        RESET,
        FAIL
};
```

Table 4-17 ControlDescriptor Enumeration Table

Name	**Purpose**
PROCESS	State transitioning and subject change by a task is authorized.
TRANSITIONAL	Subject change is not authorized. Target state transitioning is authorized.
RESET	Neither target state nor subject changes are authorized but the current state is re-entered and as such, active timeout constraints are reinitialized.
FAIL	Neither target state nor subject change are authorized. No timeout change occurs.

4.3.6 CompoundTransition

CompoundTransition is a specialization of **Transition** that introduces a **secondary** destination **State,** and **ControlDescriptor** constraint, and a reference to an **EncounterTemplate** and default initialization **Transition**.

During the invocation of the apply operation under **Collaboration** an implementation is responsible to instantiating a process described under the **template** declaration. A transition of this type will launch an **Encounter** or series of **Encounter** instances as

sub-processes to the active **Collaboration**. Determination of the selection of the **primary** destination or **alternate** destination is a function of the **result** status event raised by the transitioning **Encounter**.

CompoundTransition provides a powerful mechanism to express recursive collaborative processes such as amendments under multilateral negotiation.

4.3.6.1 IDL Specification

```
interface CompoundTransition :
        Transition
        {
        readonly attribute State secondary_target;
        readonly attribute ControlDescriptor secondary_control;
        readonly attribute EncounterTemplate template;
        readonly attribute Transition initialization;
        readonly attribute boolean sealed;
};
```

Table 4-18 CompoundTransition Attribute Table

Name	Type	Properties	Purpose
secondary_target	State	read-only	Declaration of a **State** that constitutes the alternative **State** destination to the principal destination inherited from **Transition**.
secondary_control	ControlDescriptor	read-only	Control descriptor that qualifies subject modification rights under the secondary destination.
initialization	Transition	read-only	The default initialization transition to be invoked from the possible initialization transitions exposed by the **EncounterTemplate** referenced by the **template** attribute.
sealed	boolean	read-only	Controls the exposure of a template under the template attribute. Sealed transitions will return a null to a client on an attempt to navigate to the associated template.
template	EncounterTemplate	read-only	A Template defining a process to be executed and concluded under a **success** or **failed** state. An instance of template is not exposed if the value of **sealed** is true.

4.4 Engagement and Associated Interfaces

EngagementTemplate, **Engagement**, and **EngagementManifest** are a set of interfaces used to establish, execute, and persistently register the result of a contractual engagement. **EngagementTemplate** exposes a factory key used by a client to establish an **Engagement** process and associates an instance of **EngagementManifest** as the resource produced by the **Engagement** process.

4.4.1 Object Model

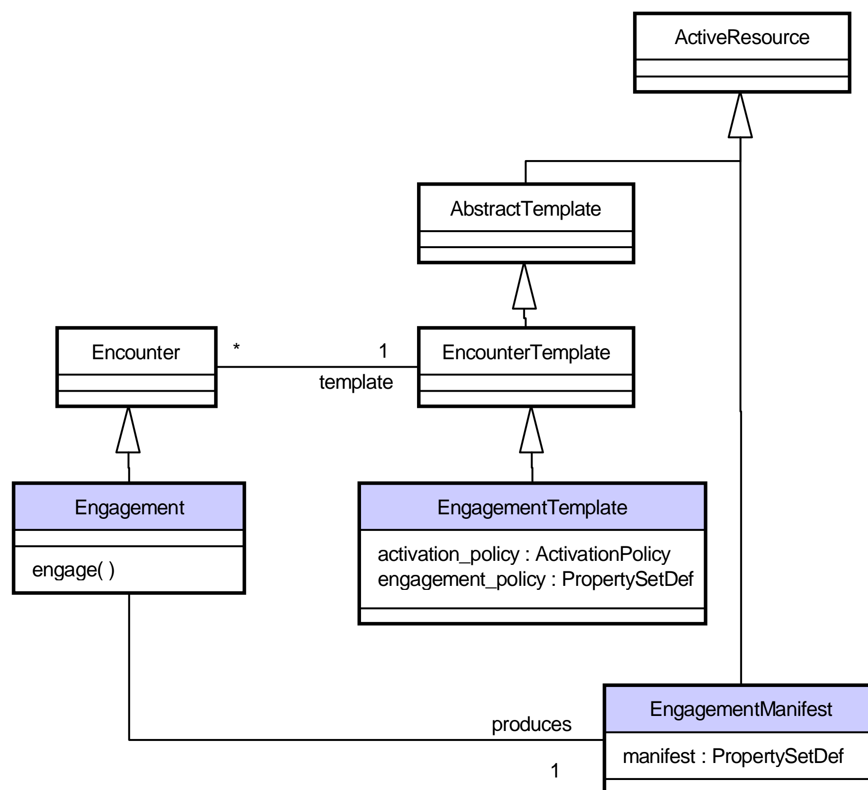

Figure 4-5 Engagement Object Model

4.4.2 EngagementTemplate

EngagementTemplate defines the criteria under which an **Engagement** is executed through the exposure of an **activation_policy** and **engagement_policy**. **EngagementTemplate** is associated to an **Engagement** process through the **data** usage association.

4.4.2.1 IDL Specification

```
interface EngagementTemplate :
    EncounterTemplate
    {
    enum ActivationPolicy{
        DISCRETIONARY,
        IMPLICIT
    };
    readonly attribute ActivationPolicy activation_policy;
    readonly attribute CosPropertyService::PropertySetDef
                engagement_policy;
};
```

Table 4-19 EngagementTemplate Attribute Table

Name	Type	Properties	Purpose
activation_policy	ActivationPolicy	read-only	An **ActivationPolicy** is one of the enumerated values **DISCRETIONARY** or **IMPLICIT** (see Table 4-21 on page 4-26) .
engagement_policy	PropertySetDef	read-only	A property set used to disclose the non-repudiation policy applicable to the engagement process. Property names and values are undefined and subject to resolution by negotiation between parties.

Table 4-20 Engagement Template Attribute Table

Name	Type	Properties	Purpose
activation_policy	ActivationPolicy	read-only	An **ActivationPolicy** is one of the enumerated values **DISCRETIONARY** or **IMPLICIT** (see Table 4-21 on page 4-26).
engagement_policy	PropertySetDef	read-only	A property set used to disclose the non-repudiation policy applicable to the engagement process. Property names and values are undefined and subject to resolution by negotiation between parties.

Table 4-21 ActivationPolicy Enumeration Table

Value	Description
DISCRETIONARY	Engagement is considered complete at the discretion of the implementation. Examples of discretionary engagement include open contracts under which participants may choose to engage. Typically, a discretionary engagement will be defined as a timeout transition such that the set of engaged parties at the point of timeout constitute the contracting parties.
IMPLICIT	Implicit engagement is defined as an engagement process that requires the explicit engagement of all participants associated to the Membership that the Collaboration represents.

4.4.3 Engagement

Engagement is a specialization of **Encounter** that associates an **EngagementTemplate** (as qualifying criteria for the engagement process) with a produced **EngagementManifest**. **Engagement** exposes the operation **engage** that takes **evidence** as input and provides **proof** of engagement as an output argument. An implementation of **Engagement** registers the set of engagement proofs under an **EngagementManifest**. The **proof** and **evidence** arguments to engage are defined by the **engagement_policy** exposed by **EngagementTemplate**.

4.4.3.1 IDL Specification

```
interface Engagement :
        Encounter
        {
        void engage(
            in any evidence,
            out any proof
        );
};
```

4.4.4 EngagementManifest

An **EngagementManifest** is a specialization of **ActiveResource** and **Descriptive** that provides a persistent store for the registration of proofs to engagement by an **Engagement** process under the **property_set** exposed by the inherited Descriptive interface. The semantics of proof registration and format are defined by an instance of **EngagementTemplate**.

4.4.4.1 IDL Specification

```
interface EngagementManifest :
        SessionFramework::ActiveResource
```

```
        {
        readonly attribute CosPropertyService::PropertySetDef manifest;
};
```

Table 4-22 EngagementManifest Attribute Table

Name	Type	Properties	Purpose
manifest	PropertySetDef	read-only	A property set used to separate evidence that may be used in an engagement process as defined by the active engagement policy. Usage is policy dependent.

4.5 Voting and Associated Interfaces

Voting is a specialization of **Encounter** that associates **VoteTemplate** (as qualifying criteria for the **Voting** process) with a produced **VoteManifest**. **Voting** exposes the **vote** operation that takes one of the enumerated value **YES**, **NO**, or **ABSTAIN** as an input argument. An implementation of **Voting** is responsible for the updating of the voting status under the **VoteManifest**.

4.5.1 Object Model

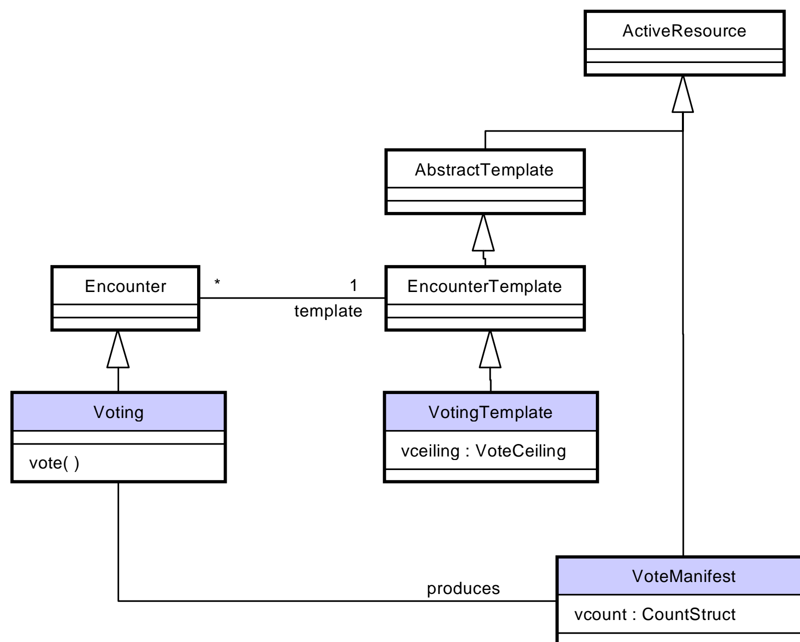

Figure 4-6 Voting Object Model

4.5.2 VoteTemplate

4.5.2.1 IDL Specification

```
interface VoteTemplate :
        Encounter
        {
        struct VoteCeiling{
            short numerator;
            short denominator;
        };
        readonly attribute VoteCeiling vceiling;
};
```

Table 4-23 VoteTemplate Attribute Table

Name	Type	Properties	Purpose
vceiling	VoteCeiling	read-only	The ceiling exposes a fractional value indicating the proportion of YES votes required to conclude a vote process successfully. Values of ceiling such as ½ or ¾ are expressed by the VoteCeiling structure in the form of a numerator and denominator value.

Table 4-24 VoteCeiling Struct Table

Element	Type	Description
numerator	short	Value describing the numerator of a fraction that in combination with the denominator defines the fractional value attributed VoteCeiling.
denominator	short	Value describing the denominator of a fraction that in combination with the numerator defines the fractional value attributed VoteCeiling.

4.5.3 Voting

Voting is a specialization of **Encounter** supporting vote-based determination of encounter success or failure. **Voting** is an interface that provides mechanisms through which users can register **YES**, **NO**, or **ABSTAIN** votes in accordance with the process policy exposed by **VoteTemplate**. **VoteTemplate** exposes a structured numerator/denominator pair that defines the required ceiling for calculation of a successful vote. An implementation of **Voting** is required to raise a **result** event on completion, indicating the successful or unsuccessful conclusion of the engagement process.

4.5.3.1 IDL Specification

```
interface Voting:
        Encounter
        {
        enum VoteDescriptor{
            YES,
            NO,
            ABSTAIN
        };
        void vote(
            in VoteDescriptor value
        );
};
```

Table 4-25 VoteDescriptor Enumeration Table

Element	Description
YES	Value signifying an affirmative position under a vote operation.
NO	Value signifying an opposing position under a vote operation.
ABSTAIN	Value signifying neither an affirmative nor negative position under a vote operation.

4.5.3.2 Registering a Vote

Votes are registered against a **Voting** process using the **vote** operation. The input argument to the **vote** operation is one of the enumerated values **YES**, **NO**, or **ABSTAIN**. A user may invoke a vote more than one time, the last vote registered is recorded as the standing vote for that **Member**.

4.5.4 VoteManifest

A **VoteManifest** is a container for the persistent registration of voting results.

The attribute count on **VoteManifest** provides an aggregation of votes registered under an active voting process. As votes are registered the values attributed to **VoteCount** are updated. As **VoteManifest** is itself an **ActiveResource**, changes to the value of **count** are raised as **update** events.

4.5.4.1 IDL Specification

```
interface VoteManifest :
    SessionFramework::ActiveResource {
    struct CountStruct{
        long yes;
        long no;
        long abstain;
    };
```

```
    readonly attribute CountStruct vcount;
};
```

Table 4-26 Voting Attribute Table

Name	Type	Properties	Purpose
vcount	CountStruct	read-only	**VoteCount** is a structure containing the number of **YES**, **NO** and **ABSTAIN** votes registered under the voting process.

Table 4-27 VoteCount Struct Table

Element	Type	Description
yes	long	Number of yes votes registered under the voting process.
no	long	Number of no votes registered under the voting process.
abstain	long	Number of abstain votes registered under the voting process.

4.6 Negotiation and Promissory Models

4.6.1 Bilateral Negotiation

4.6.1.1 Overview

A **bilateral** negotiation is a collaborative process model dealing with interactions between two participants. It provides a framework within which a user can initiate a process under which agreement to the **subject** of **Collaboration** can be established though interaction with another user. The model exposes three negotiable states, **requested**, **proposed**, and **offered**, that, through collaborative interaction may lead to any of the terminal states of **agreed**, **rejected**, or **timeout.**

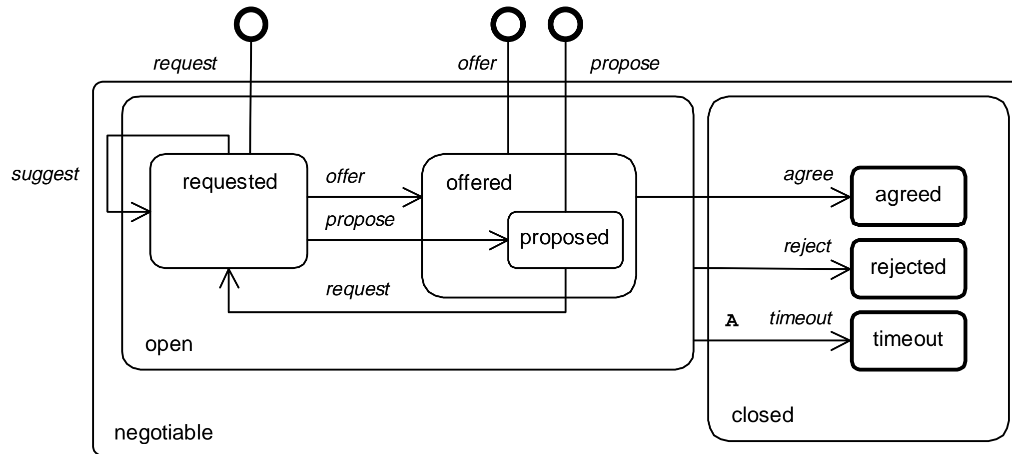

Figure 4-7 Bilateral State Transition Model

The different open sub-states of a bilateral negotiation model provide varying degrees of co-operation, commitment, and agreement. A degree of commitment is encountered under the **offered** state by the fact that an offer can be agreed to. As **proposed** is a stub-state of **offered**, it may also be agreed to; however, **proposed** extends the semantic model of **offered** by enabling the continuation of subject modification through the **request** transition.

Table 4-28 Bilateral Negotiation State Variance Table

	suggested	**requested**	**proposed**	**offered**	**agreed**	**rejected**
Expression of willingness to continue negotiation	√	√	√			
Represents commitment by the principal			√	√		
Represents a bilateral commitment					√	
Terminal state					√	√

4.6.1.2 *Initialization*

The **bilateral negotiation** may be initialized under one of the three states **proposed**, **requested**, or **offered**. An **offer** signifies a state in which the subject of collaboration may be agreed to but not be changed, whereas a **proposed** state enables the introduction of counter requests. Both **offer** and **propose** imply a potential agreement by the offering party whereas the **requested** state implies no commitment by the invoking party.

4.6.1.3 Transitions

Request

Request is a transition that can be applied under the **proposed** state. **Request** enables a respondent to change the subject of a negotiation and the context from the **proposed** to **requested** state. A **request** transition does not signify the commitment of the requesting party; however, it opens the possibility for the counterpart to respond with **propose** or **offer** against the subject under the **requested** state.

Table 4-29 Request Transition Parameter Table

keyword	mode	source	destination	priority	constraint
request	RESPONDENT	proposed	requested	0	PROCESS

Suggest

Suggest is semantically equivalent to the **request** transition except that it is initiated under the **requested** state. **Suggest** is used as an exploratory mechanism through which two clients can continue to invoke suggestions towards each other relative to the subject, until such time that at least one party is ready to migrate to a higher level of commitment as expressed under the **proposed** or **offered** states.

Table 4-30 Suggest Transition Parameter Table

keyword	mode	source	destination	priority	constraint
suggest	RESPONDENT	requested	requested	0	PROCESS

Propose

Propose is a transition from the **requested** to **proposed** states that introduces the commitment by the proposing party in that the subject of the proposal may be agreed to by the correspondent. This is distinct to the **requested** state where, in comparison, no agreement is implied.

Table 4-31 Proprose Transition Parameter Table

keyword	mode	source	destination	priority	constraint
propose	RESPONDENT	requested	proposed	0	PROCESS

Offer

An **offer** is a transition from the **requested** state to the **offered** state. Invoking **offer** is on one hand an expression of agreement by the offering party, but on the other hand, restricts the potential for further negotiation (as compared to propose).

Table 4-32 Offer Transition Parameter Table

keyword	mode	source	destination	priority	constraint
offer	RESPONDENT	requested	offered	0	PROCESS

Agree

The **agree** transition is available to a respondent under the **offered** and **proposed** states. **Agree** signifies the agreement by the respondent to an **offer** or **proposal** raised by the issuing user. The **agree** transition establishes a collaboration under an **agreed** state, expressing the agreement by both parties to the subject of a collaboration.

Table 4-33 Agree Transition Parameter Table

keyword	mode	source	destination	priority	constraint
agree	RESPONDENT	offered	agreed	0	PROCESS

Reject

A **reject** transition may be invoked against any **open** state (**proposed**, **requested**, or **offered**). **Reject** invokes a failure termination of a negotiation through transitioning to the **rejected** state.

Table 4-34 Reject Transition Parameter Table

keyword	mode	source	destination	priority	constraint
reject	RESPONDENT	open	rejected	0	PROCESS

Timeout

A **timeout** transition is associated to the **open** state and as such is active during any of the **proposed**, **requested**, or **offered** states. The timeout signifies the amount of time following the last transition which, when elapsed, will cause the execution of the transition. The result of the **timeout** transition is automatic transition to **timeout** and subsequent raising of the failure status of the host process.

Table 4-35 Timeout Transition Parameter Table

keyword	mode	source	destination	priority	constraint
timeout	TIMEOUT	open	timeout	100	TRANSITIONAL

4.6.1.4 States

The semantics of the bilateral negotiation states are summarized in the following tables.:

Table 4-36 Bilateral Negotiation State Semantics

State	terminal	Description
open	INTERNAL	The **open** state is a parent state to the three negotiable states **offered**, **proposed**, and **requested**. The three negotiable states are sub-states of the **open** state, as such transitions defined under the open state are available at any time between initialization and termination. Transitions declared on the open state enable the explicit rejection of a **subject** by a user through the **reject** transition. A second characteristic of the **open** state is the association of a **timeout** transition that will close the negotiation after a predetermined period of transition inactivity.
offered	INTERNAL	The **offered** state enables a respondent to **agree** or **reject** an agreement to the subject of the collaboration. Invoking **agree** leads to the establishment of the terminal state expressing agreement by both parties to the **subject** of the **Collaboration**.
proposed	INTERNAL	The **proposed** state extends the semantics of the **offered** state by introducing the possibility of change to the subject of the collaboration. Through application of the **request** transition, a respondent may change the subject of the collaboration to a new value and establish the active state as **requested**.
requested	INTERNAL	The requested state exposes transitions that allow a respondent to transition to the **offered** or **proposed** states using the **offer** or **propose** transitions, or to continue in the **requested** state through application of the **suggest** transition.
agreed	SUCCESS	The **agreed** state is a terminal success state that signifies the agreement of both parties to the **subject** of the **Collaboration**.
rejected	FAILURE	The **reject** state is a terminal fail state that signifies the non-agreement to the **subject** of the **Collaboration** and the termination of the process.
timeout	FAILURE	The **timeout** state is a terminal fail state that signifies the closure of the process without achievement of agreement to the **subject** of the **Collaboration**.

4.6.2 Multilateral Negotiation

4.6.2.1 Overview

A **multilateral encounter** is a collaborative process model dealing with interactions between a group of two or more participants. It provides a framework within which a user can initiate an action under which agreement to the **subject** of **Collaboration** can be established through a consensus process.

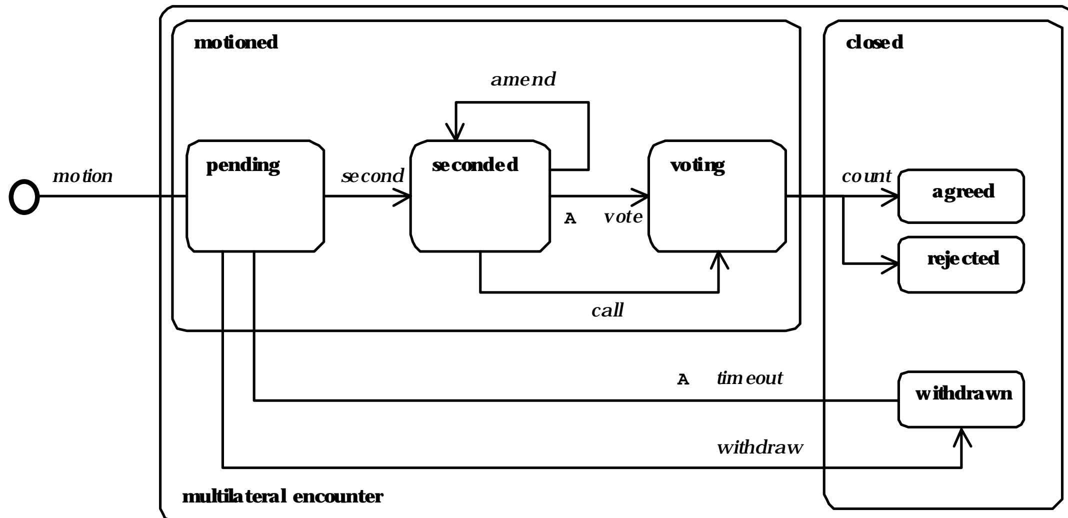

Figure 4-8 Multilateral State Transition Model

The model exposes three principal states: **pending**, **seconded**, and **voting** that through interaction between the participants may lead to any of the terminal states of **agreed**, **rejected**, or **withdrawn**. Initialization of a **multilateral encounter** is established through the initializing transition named **motion**. Under the **pending** state three possible actions are possible:

1. The user raising the motion may **withdraw** the motion,
2. a reciprocating user may **second** the motion, or
3. the motion may fail through a **timeout** due to the lack of a second.

Both **timeout** and **withdraw** transitions lead to the failed state **withdrawn**. Once a **pending** motion is **seconded** by a reciprocating user (any user other than the user raising the motion) the **vote** timeout countdown is activated. Any user may invoke the **amend** or **call** transition prior to the **vote** transition. Both **amend** and **call** transitions are executed as a sub-process defined by a **multilateral motion** (as such, both are subject to the agreement of the participants in order to succeed).

Once the **voting** state is established through a timeout of the **vote** transition, or a successful **call** transition, a **count** transition is immediately activated. The **count** transition is a specialized **Encounter** named **Voting** that exposes a **vote** operation under which participants may register **YES**, **NO**, and **ABSTAIN** votes. The success or failure of the count transition signals the success or failure of the multilateral motion process by completing the transition to either the **agreed** or **rejected** terminal states.

Table 4-37 Multilateral Negotiation State Variance Table

	pending	seconded	voting	agreed	rejected
Represents commitment by the principal	√	√	√	√	
Endorsement of a commitment by a second participant		√	√	√	
Represents a unilateral commitment				√	
Terminal state				√	√

4.6.2.2 Transitions

Motion

Initialization using **motion** establishes the **Collaboration** with the **pending** state and all parent states as the **active-state** path. A motion is raised with the express interest of gaining the agreement of the membership to the subject of the motion. For a motion to be successful, the motion must be seconded and voted upon. At any time before a motion vote is initiated the principal raising the motion may withdraw it. A potential risk of raising a motion is that the subject of the motion, if seconded, may be amended at the discretion of the group.

Table 4-38 Motion Parameter Table

keyword	mode	source	destination	priority	constraint
motion	INITIATOR		pending	0	TRANSITIONAL

Second

The **second** transition is a simple transition that may be invoked by a **respondent** in support of a **pending** motion. The second transition will result in the establishment of the **seconded** state and all **parent** states as the active-state path.

Table 4-39 Second Parameter Table

keyword	mode	source	destination	priority	constraint
second	RESPONDENT	pending	seconded	0	TRANSITIONAL

Amendment

The **amend** transition is a compound transition defined by a subsidiary collaboration process using the **multilateral motion** state model. During the invocation of the **apply** operation, the client passes in amend as the **transition** argument value, and a reference to a task that will change the current **subject**. On conclusion of the amendment process, a successful **result** will cause the completion of the transition by changing the **active-state** to **seconded** and committing the transaction. In the case of failure, no state change will occur and a rollback of changes to the **subject** will be invoked.

The expiry of the **seconded** state is re-initialized under each re-entry to the **seconded** state. As such, any amendment (successful or otherwise) will invoke the resetting of the vote transition deadline.

Table 4-40 Amend Parameter Table

Feature	Value	constraint
keyword	amend	
mode	PARTICIPANT	
template	Collaboration, using multilateral motion	
priority	0	
source	seconded	
target	seconded	PROCESS
secondary		FAIL

Call

Calling the question is a compound transition that if successful results in a transition to the **voting** state (**TRANSITIONAL** semantics).

The **call** transition is a compound transition defined by a subsidiary collaboration process using the **multilateral encounter** state model. During the invocation of the **apply** operation, the client passes in a reference to the **call** transition (task arguments are ignored). On conclusion of the **call** process (signaled by a **result** event of the **Collaboration** sub-process), a successful result will cause the completion of the transition through a change in the active state of the parent **Collaboration** to **voting**. An unsuccessful result will not invoke a transition.

Table 4-41 Call Parameter Table

Feature	Value	constraint
keyword	call	
mode	RESPONDENT	
template	Collaboration, using multilateral motion	
priority	10	
source	seconded	
target	voting	TRANSITIONAL
secondary		FAIL

Vote

The vote transition is a **TIMEOUT** transition associated to the **seconded** state. On expiry the **vote** transition is applied by **Collaboration**. The **vote** transition establishes the **voting** state.

Table 4-42 Vote Parameter Table

keyword	mode	source	destination	priority	constraint
vote	TIMEOUT	seconded	voting	100	TRANSITIONAL

Count

On activation of the voting state a **count** transition is initiated under a zero lifetime timeout (i.e., immediately on state activation). Any participant may attempt to force a vote by calling the question using the **call** transition.

The **count** transition is a compound transition defined by **Voting**. On initialization the **Voting** process is established as a sub-process of the active **Collaboration**.

Following creation of a vote aggregation sub-process, participants associated with the **Collaboration** may invoke **YES**, **NO**, or **ABSTAIN** votes using the **vote** operation on the **Voting** interface. On conclusion of the voting process as a result of a timeout of the process or registration of votes by all participants, a vote count is conducted and established under the count attribute of a **VoteManifest**. A successful conclusion of the vote process will result in a transition to the **agreed** state (**TRANSITIONAL** semantics) whereas failure will result in transition to the **rejected** state (**TRANSITIONAL** semantics).

Table 4-43 Count Transition Table

Feature	Value	constraint
keyword	count	
mode	TIMEOUT	
template	Voting	
priority	100	
source	seconded	
target	agreed	TRANSITIONAL
secondary	rejected	TRANSITIONAL

Timeout

The **timeout** transition triggers an automatic transition from the **pending** to **withdrawn** state after a set period of inactivity as disclosed by the state **model** associated to the **Collaboration** process.

Table 4-44 Timeout Parameter Table

keyword	mode	source	destination	priority	constraint
timeout	TIMEOUT	pending	withdrawn	100	TRANSITIONAL

Withdraw

The **withdraw** transition may be invoked by the principal establishing a motion at any time prior to the motion being **seconded** or the occurrence of a **timeout**. A **withdraw** transition establishes the **withdrawn** state as active, resulting in the failure of the collaboration.

Table 4-45 Withdraw Parameter Table

keyword	mode	source	destination	priority	constraint
withdraw	INITIATOR	pending	withdrawn	0	TRANSITIONAL

4.6.2.3 States

Table 4-46 Multilateral Negotiation State Semantics

State	terminal	Description
multilateral	INTERNAL	The top-level state containing the motioned and closed states.
motioned	INTERNAL	Contextual state containing the pending, seconded, and voting states.
pending	INTERNAL	The **pending** state signifies the agreement by one party to a motion, expressed as the subject of **Collaboration** and the expression of the interest of that party in the reaching of agreement to the said subject. The issuing user may withdraw a motion at any time prior to **second** transition or **timeout**. A motion fails if the **timeout** passes prior to the occurrence of a **second** transition. A **second** transition establishes the motion as a valid motion to the **Membership**.
seconded	INTERNAL	Under the **seconded** state the subject of a motion may be amended though the invocation of the **amend** transition. An **amend** transition causes the creation of a subsidiary **Collaboration** to the current **Collaboration**. Success of the subsidiary process is required before the principal **Collaboration** subject is updated. A **call** transition takes priority over any queued amendment transitions and if successful, forces a vote on the current motion. A **call** transition is executed as a subsidiary **Collaboration** using a **multilateral motion** process.
voting	INTERNAL	An immediate **timeout** of the **voting** state is triggered under the count transition. This transition creates a subsidiary **Encounter** using the **Voting** process. The boolean result of the voting process will be signaled under a **result** event that invokes completion of the transition to either the **agreed** or **rejected** state.
withdrawn	FAILURE	A state resulting from the **withdraw** of a motion prior to the occurrence of a **second** transition or a **timeout**. The **withdrawn** state signifies a failure of the multilateral encounter.
agreed	SUCCESS	An **agreed** terminal state indicating the successful resolution of the voting process by the registration of a sufficient number of **yes** votes to equal or exceed the vote **vceiling**.
rejected	FAILURE	A **rejected** terminal state indicating a failure of the voting process.

4.6.3 Promissory Encounter

4.6.3.1 Overview

The **promissory** encounter model defines a collaborative interaction sequence between a consumer and a provider. A **consumer** is a **Member** associated to a **Membership** of the kind "consumer." A **provider** is a **Member** associated to the **Membership** of the kind "provider." A **provider** can invoke a promise transition to initialize a **Collaboration** under the **right** state. Once initialized as a **right**, a consumer may call the promise by invoking a **request** transition. This corresponds to a **consumer** request for fulfillment of the promise by the **provider**. A provider fulfills a promise by applying the **fulfill** transition, itself a compound transition defined by a **bilateral** or **multilateral** negotiation. Success of the negotiation leads to the **fulfilled** state whereas failure leads to the **rejected** state.

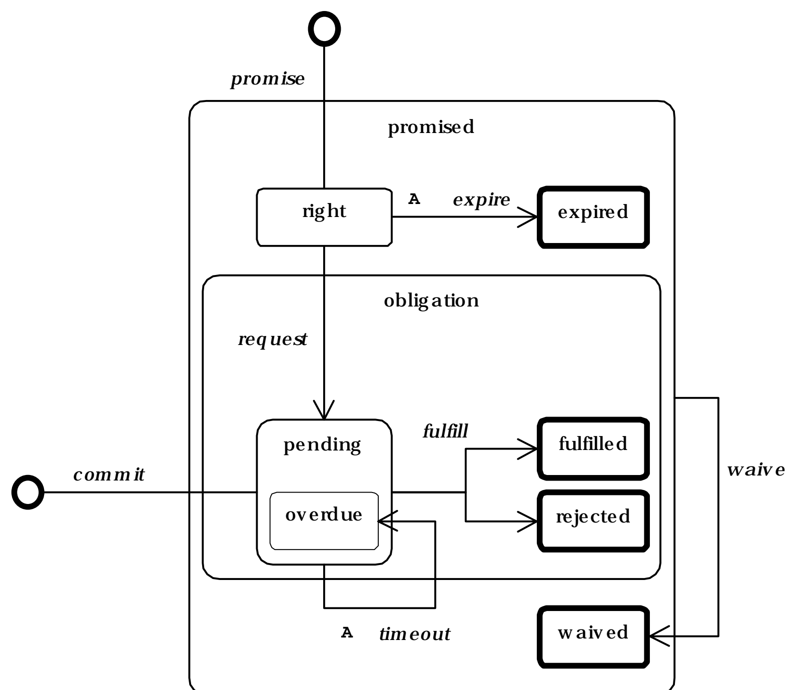

Figure 4-9 Promissory State Transition Model

4.6.3.2 Initialization

Initialization is achieved using the **promise** or **commit** transition. When initialized under **promise**, the Collaboration is established as a **right** of the consumer. When initialized under commit, the promise is established as a **pending** obligation of the provider.

Table 4-47 Promissory Initialization Table

keyword	**mode**	**kind**	**target**
promise	PARTICIPANT	provider	right
commit	PARTICIPANT	provider	pending

4.6.3.3 Transitions

Request

Request is a transition available to a consumer under the **right** state. Invoking the **request** transition establishes the promise as a **pending** obligation against the supplier.

Table 4-48 Request Transition Parameter Table

keyword	**mode**	**kind**	**priority**	**source**	**destination**	**constraint**
request	PARTICIPANT	consumer	0	right	obligation	TRANSITIONAL

Fulfill

Fulfill is available to a provider under the obligation **pending** state. A **fulfill** transition is defined as a compound transition that referenced a **bilateral** or **multilateral** negotiation template. A subsidiary **Collaboration** is instantiated that, on resolution, defines the success or failure condition used to determine a transition to the terminal **fulfilled** or **rejected** state.

Table 4-49 Fulfill Transition Parameter Table

Feature	**Value**	**constraint**
keyword	fulfill	
mode	PARTICIPANT	
role name	provider	
template	bilateral or multilateral negotiation	
priority	0	
source	pending	
target	fulfilled	PROCESS
secondary	rejected	TRANSITIONAL

Waive

Waive may be invoked by either consumer or provider. It is a compound transition referencing a bilateral or multilateral negotiation that if successful results in a transition to the terminal **waived** state. A failure of the negotiation will result in the continuation of the process under the state prior to the initiation of the **waive** transition.

Table 4-50 Waive Transition Parameter Table

Feature	Value	constraint
keyword	waive	
mode	PARTICIPANT	
role name		
template	bilateral or multilateral negotiation	
priority	0	
source	promised	
target	waived	TRANSITIONAL
secondary		FAIL

Expire

Expire exposes a timeout value that will trigger the expiry of the consumer's right to invoke a **request** for fulfillment against a provider.

Table 4-51 Expire Transition Parameter Table

keyword	mode	kind	priority	source	destination	constraint
expire	TIMEOUT		100	right	expired	TRANSITIONAL

Timeout

Timeout changes an existing **obligation pending** to **obligating pending** and **overdue**. From a computation point of view an **overdue** obligation is no different to a **pending** obligation.

Table 4-52 Timeout Transition Parameter Table

keyword	mode	kind	priority	source	destination	constraint
timeout	TIMEOUT		100	pending	overdue	TRANSITIONAL

4.6.3.4 States

Table 4-53 Promissory State Table

State	terminal	Description
promised	INTERNAL	The top level state exposing a promissory encounter model.
right	INTERNAL	A promise, made by a provider, towards a consumer under which the provider commits to the willingness to fulfill the promise at the request of the consumer.

Table 4-53 Promissory State Table

obligation	INTERNAL	A promise that has been requested by a consumer, or initialized through a commit, under which the promise constitutes an obligation of the provider to fulfill. Obligation is a contextual state that qualifies the operation states of pending, fulfilled, and rejected.
pending	INTERNAL	A state under which a provider is obliged to fulfill a promise through invocation of the fulfil transition.
overdue	INTERNAL	A sub-state of pending which is established by an implementation of Collaboration when a pending obligation timeout transition expires.
waived	SUCCESS	A sub-state of obligation, reached through mutual agreement of the parties, under which the obligations and rights of both parties are forgone.
fulfilled	SUCCESS	A success terminal state, expressing the satisfactory fulfillment of a promise by a provider towards a consumer.
rejected	FAILURE	A failure terminal state, expressing the failure of the parties to agree to the fulfillment of a promise.

4.6.4 CollaborationFramework IDL

```
// File: CollaborationFramework.idl

#ifndef _COLLABORATION_FRAMEWORK_IDL_
#define _COLLABORATION_FRAMEWORK__IDL_
#pragma prefix "omg.org"

#include <CommunityFramework.idl>

module CollaborationFramework{

    // forward declarations
        interface State;
        interface Trigger;
        interface Command;
        interface Transition;
        interface CompoundTransition;

        interface Encounter;
        interface EncounterTemplate;
        interface Implication;

        interface Collaboration;
        interface CollaborationTemplate;

        interface Engagement;
        interface EngagementTemplate;
        interface EngagementManifest;

        interface Voting;
```

```
interface VoteTemplate;
interface VoteManifest;

// type definitions

typedef string Keyword;

typedef sequence <State> StateSequence;
typedef sequence <Transition> TransitionSequence;
typedef sequence <Collaboration> CollaborationSequence;
typedef sequence <Command> CommandSequence;

// encounter and template interfaces

interface EncounterTemplate :
    SessionFramework::AbstractTemplate
    {
    readonly attribute CommunityFramework::MembershipKind
                membership_kind;
    enum ImplicationOrdering {SEQUENTIAL,
                        PARALLEL
    };
    readonly attribute ImplicationOrdering order;
};

interface Implication : SessionFramework::Linkage{ };
interface Success : Implication{ };
interface Failure : Implication{ };

interface Encounter :
    CommunityFramework::Membership,
    SessionFramework::ActiveTask
    {
    readonly attribute EncounterTemplate template;
    readonly attribute SessionFramework::ActiveResource subject;
};

// interfaces

interface State :
    Session::BaseBusinessObject{
    readonly attribute Keyword label;
    enum TerminalDescriptor{
        INTERNAL,
        SUCCESS,
        FAILURE
    };
    readonly attribute TerminalDescriptor terminal;
    readonly attribute State parent;
    readonly attribute StateSequence sub_states;
};
```

```
interface Trigger :
    Session::BaseBusinessObject{
    readonly attribute CollaborationFramework::Keyword keyword;
    enum TriggerMode{
        INITIATOR,
        RESPONDENT,
        PARTICIPANT,
        TIMEOUT
};
    readonly attribute State source;
    readonly attribute TriggerMode mode;
    readonly attribute CommunityFramework::MembershipKind
            constraint;
    readonly attribute long priority;
    readonly attribute TimeBase::IntervalT lifetime;
};

interface Command :
    CollaborationFramework::Trigger{
};

interface Transition :
    Trigger
    {
    enum ControlDescriptor{
        PROCESS,
        TRANSITIONAL,
        RESET,
        FAIL
    };
    readonly attribute State target;
    readonly attribute ControlDescriptor control;
    readonly attribute boolean initialize;
};

interface CompoundTransition :
    Transition
    {
    readonly attribute State secondary_target;
    readonly attribute ControlDescriptor secondary_control;
    readonly attribute EncounterTemplate template;
    readonly attribute Transition initialization;
    readonly attribute boolean sealed;
};

// Collaboration and template

interface CollaborationTemplate :
    EncounterTemplate,
    State
```

```
            {
            readonly attribute TransitionSequence transitions;
            readonly attribute CommandSequence commands;
        };

        interface Collaboration :
            Encounter
            {
            readonly attribute StateSequence active_state;
            struct TimeoutSequence{
                CollaborationFramework::Trigger trigger;
                TimeBase::UtcT timestamp;
            };
            readonly attribute TimeoutSequence timeout_list;
            exception InvalidTrigger{
                CollaborationFramework::Trigger trigger;
            };
            exception ApplyFailure{
                CollaborationFramework::Trigger trigger;
                SessionFramework::ActiveTask task;
            };
            enum ApplySemantics{
                REPLACEMENT,
                MODIFICATION
            };
            exception ActiveTaskTypeConflict{
                SessionFramework::ActiveResource resource;
            };
            void apply(
                in CollaborationFramework::Transition transition,
                in ApplySemantics semantic,
                in SessionFramework::ActiveResource resource
            ) raises (
                InvalidTrigger,
                ActiveTaskTypeConflict,
                ApplyFailure
            );
            void invoke(
                in CollaborationFramework::Command command,
                in SessionFramework::ActiveResource argument,
                in string reason
            ) raises (
                InvalidTrigger
            );
};

// Engagement Template, Process and Manifest

interface EngagementTemplate :
        EncounterTemplate
        {
```

```
        enum ActivationPolicy{
            DISCRETIONARY,
            IMPLICIT
        };
        readonly attribute ActivationPolicy activation_policy;
        readonly attribute CosPropertyService::PropertySetDef
                    engagement_policy;
        };

        interface Engagement :
            Encounter       {
            void engage(
                in any evidence,
                out any proof
            );
};

interface EngagementManifest :
        SessionFramework::ActiveResource
        {
        readonly attribute CosPropertyService::PropertySetDef manifest;
};

// Vote Template, Process and Manifest

interface VoteTemplate :
        EncounterTemplate
        {
        struct VoteCeiling{
            short numerator;
            short denominator;
        };
        readonly attribute VoteCeiling vceiling;
};

interface Voting:
        Encounter
        {
        enum VoteDescriptor{
            YES,
            NO,
            ABSTAIN
        };
        void vote(
                in VoteDescriptor value
            );
        };

        interface VoteManifest :
            SessionFramework::ActiveResource
            {
```

```
        struct CountStruct{
            long yes;
            long no;
            long abstain;
        };
        readonly attribute CountStruct vcount;
    };

}; // end CollaborationFramework Module

#endif // _COLLABORATION_FRAMEWORK_IDL_
```

DOM Framework 5

Contents

This chapter contains the following topics.

Topic	Page
“Overview”	5-1
“DomFramework Wrapper Interfaces”	5-1

5.1 Overview

It is a high priority to be able to apply the processes of negotiation and other forms of collaboration to subject resources exposing the W3C DOM level 1 interfaces. There are two problems that have to be dealt with in achieving this:

1. DOM Specification issues:
   - Illegal IDL declarations concerning exceptions raised by attributes
   - Non-support for OMG Language Mappings
   - Implied locality restrictions
2. Framework to DOM Issues:
   - Definition of interfaces enabling the representation of a DOM as a type of ActiveResource.

5.2 DomFramework Wrapper Interfaces

Resolution of the DOM Specification issues identified above has been achieved through a set of wrapper interfaces defined under the module **DomFramework**.

The wrapper interfaces introduce the following additional features:

1. **DocFramework::Node**
   - Addition of **CosObjectIdentity::IdentifiableObject**
   - Addition of a mode attribute containing **CosPropertyService::PropertyModeType** access constrain declaration
   - The addition of the **get_nodeValue** operation and exception
2. **DocFramework::CharacterData**
   - Addition of a **set_data** and **get_data** operations with exceptions
3. **DocFramework::ProcessingInstruction**
   - Addition of a **set_data** operation with exception
4. Wrapping of all interfaces to inherit from **DomFramework::Node** or its derived interface and their counterpart in the W3C DOM module

Specification of the DOM interfaces are detailed under the W3C DOM Level 1 Recommendation. Relevant W3C DOM documentation is available under the following URLs.

- W3C DOM Level 1 Recommendation http://www.w3.org/TR/REC-DOM-Level-1/
- DOM IDL http://www.w3.org/DOM/updates/REC-DOM-Level-1-19981001-errata.html or directly under the archive http://www.w3.org/DOM/updates/REC-DOM-Level-1-java-binding-19990107.zip.

Semantics of the access constraints introduced under the **DomFramework::Node** interface are detailed under the OMG **CosPropertyService** specification.

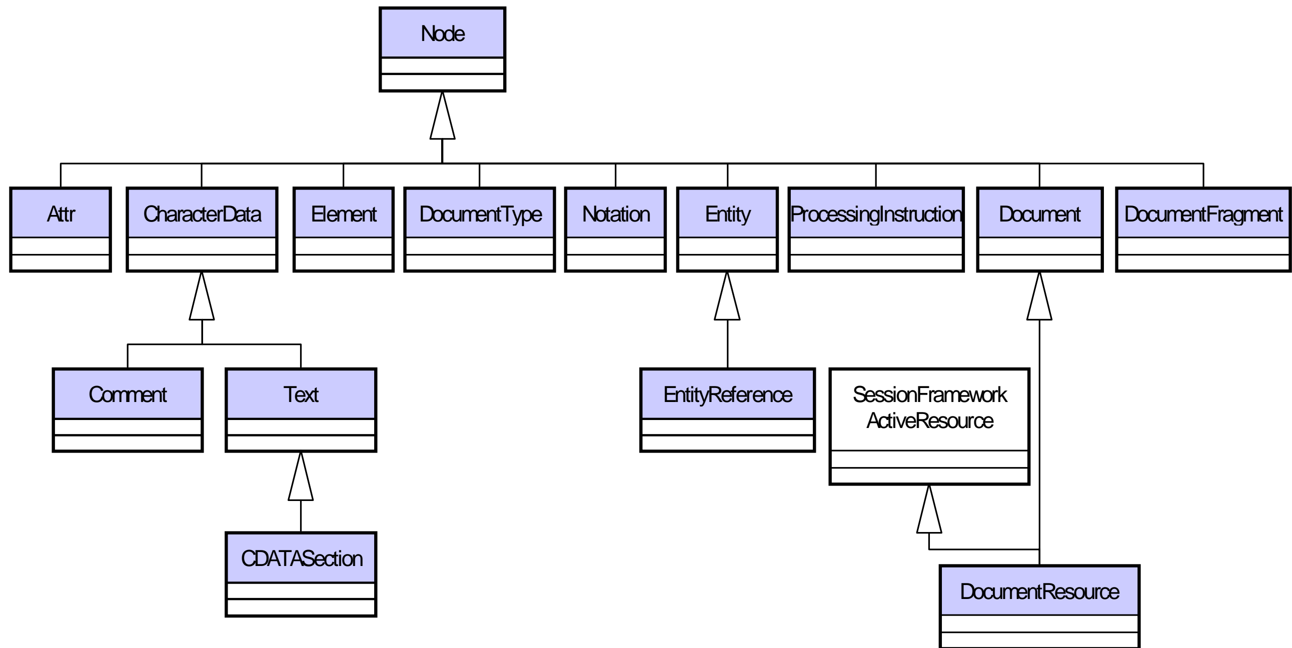

Figure 5-1 DOM wrapper interfaces defined under the DomFramework module

5.2.1 Extensions

DocumentResource has been defined to support the mapping of a **DomFramework::Document** as an **AbstractResource**.

5.2.1.1 DocumentResource

DocumentResource is derived from **DomFramework::Document** and **ActiveResource**. As an **ActiveResource**, the interface inherits life-cycle operations, which are undefined in the W3C DOM Level 1 recommendation. A Document interface represents an entire HTML or XML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data.

Since elements, text nodes, comments and processing instructions cannot exist outside the context of a Document, the Document interface also contains the factory methods needed to create these objects. The **Node** objects created have an **ownerDocument** attribute, which associates the **DocumentResource** within whose context they were created.

5.2.1.2 Object Model

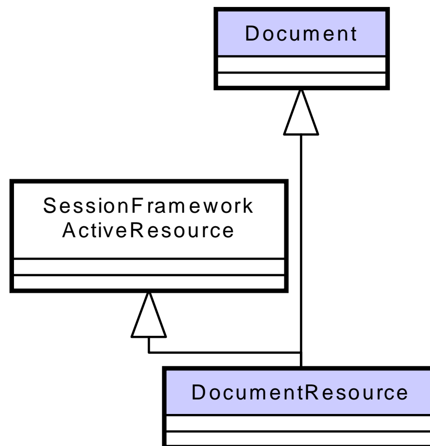

Figure 5-2 DocumentResource Object Model

5.2.1.3 IDL Specification

```
interface DocumentResource :
      DomFramework::Document,
      SessionFramework::ActiveResource{
};
```

5.2.2 DomFramework IDL

```
// File: DomFramework.idl

#ifndef _DOM_FRAMEWORK_IDL_
#define _DOM_FRAMEWORK_IDL_
#include <dom.idl>
#include <SessionFramework.idl>
#pragma prefix "omg.org"

module DomFramework {

        // forward declarations for DOM wrappers

        interface DocumentFragment;
        interface Document;
        interface Node;
        interface CharacterData;
        interface Attr;
        interface Element;
        interface Text;
```

```
interface Comment;
interface CDATASection;
interface DocumentType;
interface Notation;
interface Entity;
interface EntityReference;
interface ProcessingInstruction;

// forward declarations for extensions

interface DocumentResource;

// typedefs

typedef dom::DOMImplementation DOMImplementation ;
typedef dom::NodeList NodeList;
typedef dom::NamedNodeMap NamedNodeMap;
typedef dom::DOMString DOMString;

// dom interface wrappers

interface Node :
    CosObjectIdentity::IdentifiableObject,
    dom::Node
    {
    readonly attribute CosPropertyService::PropertyModeType
            mode;
    DOMString get_nodeValue(
    ) raises (
        dom::DOMException
    );
    void set_nodeValue( ) raises (
        dom::DOMException
    );
};

interface CharacterData :
    DomFramework::Node,
    dom::CharacterData
    {
    DOMString get_data( ) raises (
        dom::DOMException
    );
    void set_data(
        in DOMString data
    ) raises (
        dom::DOMException
    );
};

interface Attr :
```

```
        DomFramework::Node,
        dom::Attr {
};

interface Element :
        DomFramework::Node,
        dom::Element {
};

interface Text :
        DomFramework::CharacterData,
        dom::Text {
};

interface Comment :
        DomFramework::CharacterData,
        dom::Comment {
};

interface CDATASection :
        DomFramework::Text,
        dom::CDATASection {
};

interface DocumentType :
        DomFramework::Node,
        dom::DocumentType{
};

interface Notation :
        DomFramework::Node,
        dom::Notation{
};

interface Entity :
        DomFramework::Node,
        dom::Entity{
};

interface EntityReference :
        DomFramework::Node,
        dom::EntityReference{
};

interface ProcessingInstruction :
        DomFramework::Node,
        dom::ProcessingInstruction
        {
        void set_data(
                in DOMString data
        ) raises (
```

```
            dom::DOMException
        );
    };

    interface DocumentFragment :
        DomFramework::Node,
        dom::DocumentFragment{
    };

    interface Document :
        DomFramework::Node,
        dom::Document{
    };

    // Session Framework interface extensions

    interface DocumentResource :
        DomFramework::Document,
        SessionFramework::ActiveResource{
    };
};

#endif // _DOM_FRAMEWORK_IDL_
```

5.2.3 DOM Level 1 IDL (errata version)

The following IDL is provided for reference purposes only.

```
// File: dom.idl

/*
 * Copyright (c) 1998 World Wide Web Consortium, (Massachusetts Institute
 * of Technology, Institut National de Recherche en Informatique et en
 * Automatique, Keio University).
 * All Rights Reserved. http://www.w3.org/Consortium/Legal/
 */

#ifndef _DOM_IDL_
#define _DOM_IDL_
#pragma prefix "w3c.org"

module dom{

typedef sequence<unsigned short> DOMString;

    interface NodeList;
    interface NamedNodeMap;
    interface Document;

    exception DOMException {
```

```
        unsigned short code;
    };

    // ExceptionCode
    const unsigned short INDEX_SIZE_ERR = 1;
    const unsigned short DOMSTRING_SIZE_ERR = 2;
    const unsigned short HIERARCHY_REQUEST_ERR = 3;
    const unsigned short WRONG_DOCUMENT_ERR = 4;
    const unsigned short INVALID_CHARACTER_ERR = 5;
    const unsigned short NO_DATA_ALLOWED_ERR = 6;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 7;
    const unsigned short NOT_FOUND_ERR = 8;
    const unsigned short NOT_SUPPORTED_ERR = 9;
    const unsigned short INUSE_ATTRIBUTE_ERR = 10;


interface DOMImplementation {
    boolean hasFeature(
        in DOMString feature,
        in DOMString version
    );
};
    interface Node {
// NodeType
    const unsigned short ELEMENT_NODE = 1;
    const unsigned short ATTRIBUTE_NODE = 2;
    const unsigned short TEXT_NODE = 3;
    const unsigned short CDATA_SECTION_NODE = 4;
    const unsigned short ENTITY_REFERENCE_NODE = 5;
    const unsigned short ENTITY_NODE = 6;
    const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
    const unsigned short COMMENT_NODE = 8;
    const unsigned short DOCUMENT_NODE = 9;
    const unsigned short DOCUMENT_TYPE_NODE = 10;
    const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
    const unsigned short NOTATION_NODE = 12;

    readonly attribute DOMString nodeName;
    attribute DOMString nodeValue;
        // raises(DOMException) on setting
        // raises(DOMException) on retrieval
    readonly attribute unsigned short nodeType;
    readonly attribute Node parentNode;
    readonly attribute NodeList childNodes;
    readonly attribute Node firstChild;
    readonly attribute Node lastChild;
    readonly attribute Node previousSibling;
    readonly attribute Node nextSibling;
    readonly attribute NamedNodeMap attributes;
    readonly attribute Document ownerDocument;
    Node insertBefore(
```

```
        in Node newChild,
        in Node refChild
    ) raises (
        DOMException
    );
    Node replaceChild(
        in Node newChild,
        in Node oldChild
    ) raises (
        DOMException
    );
    Node removeChild(
        in Node oldChild
    ) raises (
        DOMException
    );
    Node appendChild(
        in Node newChild
    ) raises (
        DOMException
    );
    boolean hasChildNodes();
    Node cloneNode(
        in boolean deep
    );
};

interface NodeList {
Node item(
    in unsigned long index
);
readonly attribute unsigned long length;
};

interface NamedNodeMap {
Node getNamedItem(
    in DOMString name
);
    Node setNamedItem(
        in Node arg
    ) raises (
        DOMException
    );
    Node removeNamedItem(
        in DOMString name
    ) raises (
        DOMException
    );
    Node item(
        in unsigned long index
    );
```

```
        readonly attribute unsigned long length;
};

interface CharacterData : Node {
        attribute DOMString data;
                // raises(DOMException) on setting
                // raises(DOMException) on retrieval
        readonly attribute unsigned long length;
        DOMString substringData(
                in unsigned long offset,
                        in unsigned long count
        ) raises (
                DOMException
        );
        void appendData(
                in DOMString arg
        ) raises (
                DOMException
        );
        void insertData(
                in unsigned long offset,
                        in DOMString arg
        ) raises (
                DOMException
        );
        void deleteData(
                in unsigned long offset,
                        in unsigned long count
        ) raises (
                DOMException
        );
        void replaceData(
                in unsigned long offset,
                        in unsigned long count,
                        in DOMString arg
        ) raises (
                DOMException
        );
};

interface Attr : Node {
        readonly attribute DOMString name;
        readonly attribute boolean specified;
        attribute DOMString value;
};

interface Element : Node {
        readonly attribute DOMString tagName;
        DOMString getAttribute(in DOMString name);
        void setAttribute(
                in DOMString name,
```

```
            in DOMString value
        ) raises (
            DOMException
        );
        void removeAttribute(
            in DOMString name
        ) raises (
            DOMException
        );
        Attr getAttributeNode(
            in DOMString name
        );
        Attr setAttributeNode(
            in Attr newAttr
        ) raises (
            DOMException
        );
        Attr removeAttributeNode(
            in Attr oldAttr
        ) raises (
            DOMException
        );
        NodeList getElementsByTagName(
            in DOMString name
        );
        void normalize();
    };

    interface Text : CharacterData {
        Text splitText(
            in unsigned long offset
        ) raises (
            DOMException
        );
    };

    interface Comment : CharacterData { };

    interface CDATASection : Text { };

    interface DocumentType : Node {
        readonly attribute DOMString name;
        readonly attribute NamedNodeMap entities;
        readonly attribute NamedNodeMap notations;
    };

    interface Notation : Node {
        readonly attribute DOMString publicId;
        readonly attribute DOMString systemId;
    };
```

```
interface Entity : Node {
    readonly attribute DOMString publicId;
    readonly attribute DOMString systemId;
    readonly attribute DOMString notationName;
};

interface EntityReference : Node {
};

interface ProcessingInstruction : Node {
    readonly attribute DOMString target;
    attribute DOMString data;
            // raises(DOMException) on setting
};

interface DocumentFragment : Node { };

interface Document : Node {
    readonly attribute DocumentType doctype;
    readonly attribute DOMImplementation implementation;
    readonly attribute Element documentElement;
    Element createElement(
        in DOMString tagName
    ) raises (
        DOMException
    );
    DocumentFragment createDocumentFragment();
    Text createTextNode(
        in DOMString data
    );
    Comment createComment(
        in DOMString data
    );
    CDATASection createCDATASection(
        in DOMString data
    ) raises (
        DOMException
    );
    ProcessingInstruction createProcessingInstruction(
        in DOMString target,
            in DOMString data
    ) raises (
        DOMException
    );
    Attr createAttribute(
        in DOMString name
    ) raises (
        DOMException
    );
        EntityReference createEntityReference(
            in DOMString name
```

```
            ) raises (
                DOMException
            );
            NodeList getElementsByTagName(
                in DOMString tagname
            );
        };
};

#endif // _DOM_IDL_
```

Glossary A

A.1 Terms and Definitions

Term	Definition
AbstractTemplate	AbstractTemplate is an ActiveResource that exposes a factory_key and criteria. AbstractTemplate is the base type for a set of EncounterTemplate and MembershipKind.
ActiveResource	ActiveResource is a specialization of Session::AbstractResource that includes inheritance from the CosNotifyComm StructuredPushSupplier and StructuredPushConsumer interfaces. This extension introduces the ability of an abstract resource to expose structured events it is capable of producing and to subscribe to events on a selective basis. Other extensions include operations associated with the binding and release of Linkage association.
ActiveTask	ActiveTask extends Session::Task through the addition of ActiveResource and serves as a base type for Encounter.
ActiveUser	ActiveUser extends Session::User through the addition of the CosLifeCycle::FactoryFinder interface and LegalEntity. As a LegalEntity, an ActiveUser exposes public credentials that may be used under contractual engagement processes.
ActiveWorkspace	ActiveWorkspace extends Session::Workspace through ActiveResource and provides a base type for Community.
Agency	A specialization of Community and LegalEntity that introduces the notion of legal community such as a company that maintains jurisdiction of a set of resources. Agency, through LegalEntity and Jurisdiction enables the qualification of the authority of a Member within a negotiation or other collaborative encounter.

Bilateral	A bilateral negotiation is a collaborative process model dealing with interactions between two participants. It provides a framework within which a user can initiate a process under which agreement to the subject of Collaboration can be established though interaction with another user. The model exposes three negotiable states (requested, proposed and offered) that through collaborative interaction may lead to any of the terminal states of agreed, rejected, or timeout.
Collaboration	A type of Encounter bound to a CollaborationTemplate that mediates access to a subject. Collaboration exposes the state of a collaborative process and brings together the operations that may be applied by collaborating users relative to a process template. An apply operation enables the invocation of simple and compound transitions that under the mediated control of the Collaboration enable parties to reach terminal success or failure states. Users are associated to a Collaboration though a Member role.
CollaborationTemplate	CollaborationTemplate is a specialization of a State and EncounterTemplate that exposes a set of transition declarations that may be applied to an instance of Collaboration. As a State, a CollaborationTemplate exposes a sub-state hierarchy that enables the activation of command events and transition. Transitions exposed by CollaborationTemplate are declarations of source and destination states that may be used as arguments under the Collaboration interface apply operation.
Command	A specialization of Trigger that enables the declaration of an event that may be invoked under Collaboration.
Community	A specialization of ActiveWorkspace, Membership, and FactoryFinder. As an ActiveWorkspace, a Community is a place containing ActiveResources. As a Membership, a Community exposes policy concerning membership and the association of MembershipKind hierarchies. As a FactoryFinder, Community represents a possible target under a copy or move operation.
Composition	An association that signifies the composition of a target resource within a source composite resource.
Compound Transition	A specialization of Transition that introduces an alternative destination State and template describing the criteria for Encounter creation. CompoundTransition provides a powerful mechanism to express recursive collaborative encounters such as amendments under multilateral negotiation.
Containment	An association equivalent to the Task/Session Containment interface that associates a containing ActiveWorkspace with the contained ActiveResource.
Delegation	A role based Linkage that requires a concrete base type that inherits from the target type and delegates target operation to the target instance.

Desktop	SessionFramework::Desktop extends Session::Desktop and ActiveWorkspace defining an event enhanced equivalent of the Task/Session Desktop.
Document Object Model	This W3C DOM specification defines the Document Object Model Level 1, an interface that allows programs and scripts to dynamically access and update content, structure and style of XML documents.
DocumentResource	DocumentResource is derived from DomFramework::Document and ActiveResource. As an ActiveResource, the interface inherits life-cycle operations, which are undefined in the W3C DOM Level 1 recommendation. A Document interface represents an entire HTML or XML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data.
DOM	Document Object Model
Encounter	A specialization of ActiveTask and Membership that has an association to an EncounterTemplate that defines the encounter constraints, and an associated subject.
EncounterTemplate	A specialization of AbstractTemplate that references a MembershipKind applicable to an Encounter of the type described by EncounterTemplate.
Engagement	A type of Encounter defined by an associated EngagementTemplate that enables the association of proof of engagement to an agreement.
EngagementManifest	EngagementManifest is a type supporting the registration of proof as defined by the EngagementTemplate.
EngagementTemplate	Features associated to EngagementTemplate define the criteria to be applied during the engagement process.
Implication	A base type for the Success and Failure Implication linkage that associates a source template with a target template.
Jurisdiction	Jurisdiction is a specialization of the Linkage that infers authority of a LegalEntity over a resource.
LegalEntity	A type exposing a set of AbstractTemplate instances that defines a key and criteria for access to public credentials. A LegalEntity may be associated to an arbitrary number of ActiveResource instances through a Jurisdiction linkage.
Linkage	Abstract base interface that exposes a source and target of an association.
Member	A role of ActiveUser, defined as a specialization of Linkage that associates a target ActiveUser with a Membership. As a Membership may be a hierarchy of Membership instances, an instance of Member may be associated as a member at many levels within the hierarchy.

Membership	A specialization of ActiveResource that enables association of instances of the type Member in accordance with rules exposed under a MembershipKind. A Membership exposes interfaces through which Member instances may be added, removed and listed relative to the kind of participation exposed by a MembershipKind hierarchy
MembershipKind	Definition of constraints for a given MembershipKind. Constraints include the maximum number of members that may be associated under the kind, quorum value indicating the number of members that kind that must be associated and connected before the Member is considered valid, privacy policy declarations, and policies concerning the semantics of membership hierarchy.
Multilateral	A multilateral encounter is a collaborative process model dealing with interactions between a group of two or more participants. It provides a framework within which a user can initiate an action under which agreement to the subject of Collaboration can be established through a consensus process.
Model	A feature of Membership that references the root MembershipKind it is associated with. More generally, model refers to a specialization of AbstractTemplate that qualifies semantics and constraints of a process such as Encounter.
Promissory	The promissory encounter model defines a collaborative interaction sequence between a consumer and a provider. A consumer is a Member associated to a Membership of the kind “consumer.” A provider is a Member associated to the Membership of the kind “provider.” A provider can invoke a promise transition to initialize a Collaboration under the right state. Once initialized as a right, a consumer may call the promise by invoking a request transition. This corresponds to a consumer request for fulfillment of the promise by the provider. A provider fulfills a promise by applying the fulfill transition, itself a compound transition defined by a bilateral or multilateral negotiation. Success of the negotiation leads to the fulfilled state whereas failure leads to the rejected state.
Promise	The top-level state within a Promissory Encounter. Refer to promissory.
Role	Refer to Delegation.
State	A type that exposes a label, characteristics that qualify the state as internal, terminal success or terminal failure, exposes a set of sub-states, and parent state.
Subject	A reference to an ActiveResource held by an Encounter type. An Encounter mediates control over the access and modification of a subject.

Trigger	A type that exposes a keyword, accesses and timeout constraints. Triggers are used as a super-type for the Command and Transition types. Operational qualifiers include a usage mode and references to a MembershipKind that is authorized to invoke a Trigger. Usage mode enables the declaration of constraints over activation relative to the collaborative context.
Transition	A Transition extends Trigger to include a destination state. A transition may only be invoked when the active_state of collaboration is the source state in the Transition declaration. Following a successful activation of a transition, the destination state and all parents of the destination state are considered active by the controlling Collaboration.
Usage	An association equivalent to the Task/Session Usage that associates a using ActiveTask with the used ActiveResource.
Voting	A type of Encounter launched by a compound transition supporting vote-based determination of primary or alternate state selection. Voting is an interface that provides mechanisms through which users in a collaborative process can register a YES, NO, or ABSTAIN votes.
Vote	An operation available under the Voting type enabling the registration of YES, NO, and ABSTAIN votes.
VoteTemplate	VoteTemplate exposes policies concerning quorum and structured numerator/denominator pair that defines the required ceiling for calculation of a successful vote.
VoteManifest	A persistent store created by a Voting Encounter for the registration of vote results.

Object Model B

B.1 Overview

An **ActiveUser** is associated to a collaboration process through a **Member** role. A set of **Member** instances are associated together under a **Membership**. A specialization of **Membership**, called **Encounter**, extends this model to introduce an association to a defining process **template** and **subject**. **Collaboration**, **Voting**, and **Engagement** are examples of specialization of **Encounter**. A minimal client application invokes operations against a **Collaboration** instance by the passing references to tasks or resources as arguments that define actions to be applied to the **subject** of the **Encounter**. These actions are coordinated by a **Collaboration** instance in accordance with policies and constraints defined within the associated **template**. **Collaboration** mediates multiple client requests by coordinating the association of client tasks as producer of the **subject** of the mediation. As a specialization of **Encounter**, **Collaboration** has an explicit association to an **owner**, exposes relationships to consumed **process** and **data** resources, and the resources it produces. **Encounter** may expose an ordered hierarchy of sub-processes that collectively describes the state of a collaborative encounter.

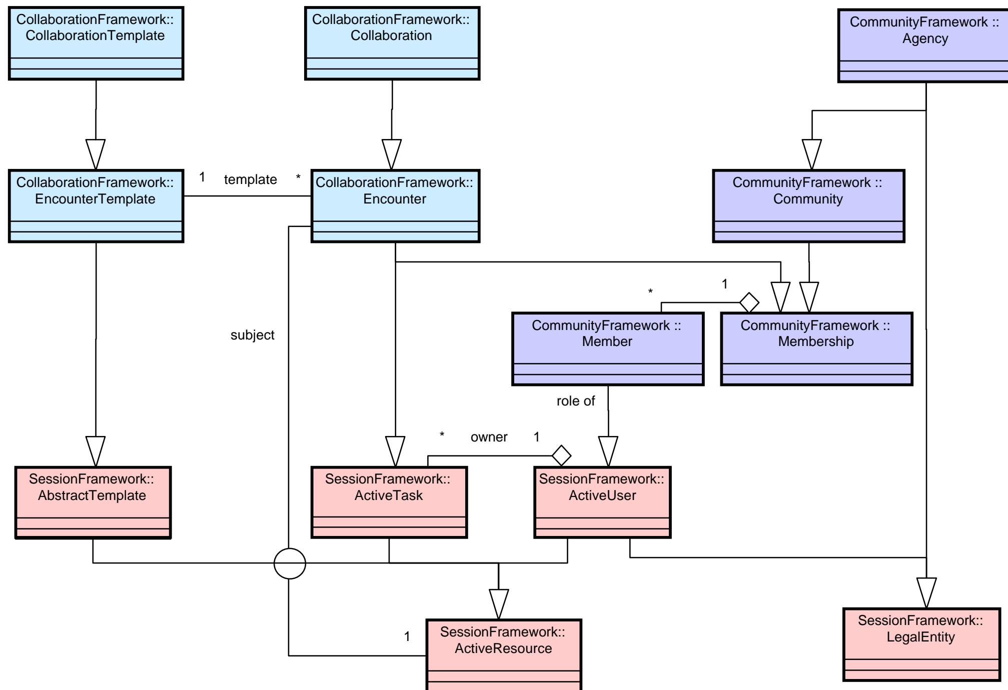

Figure B-1 Object Model Overview

The **CommunityFramework** introduces a set of interfaces supporting higher level business-to-business encounters in which users interact in the context of communities under which the role and jurisdiction of participants are exposed. Roles and the membership **kind** they represent enable the introduction of constraints into collaboration models that allows the definition of more complex collaboration processes.

During the course of a collaborative encounter, information requests may be directed to clients concerning the disclosure of restricted information. In these cases the context of a collaborative encounter is crucial to a client's determination of the appropriate disclosure policy to apply. Resolution of context is established through the semantics of the collaboration template (how), subject (what), membership (who), community (where), and collaboration process itself (state). Operations supporting domain and context dependent disclosure of information are based on **FactoryFinder** interfaces exposed under the **ActiveUser** and **Community**. This specification assumes the semantics of **create** under a **GenericFactory** that may return new or existing object references.**[Reviewer, added the word "that" please verify.]**

B.1.1 Collaboration Model

Collaborative process models are defined under the type **CollaborationTemplate**. A **CollaborationTemplate** is a specialization of a **State** and **EncounterTemplate**. It exposes a set of transitions that may be applied under an instance of **Collaboration**. As a **State**, a **CollaborationTemplate** exposes a sub-state hierarchy and a set of

command descriptors that enable the activation of command events. Initialization transitions enable the configuration of the initial active state of collaboration. Transitions exposed by **CollaborationTemplate** are declarations of source and destination states and activation constraints. **Transition** inherits activation constraints from the super-type **Trigger**. **Trigger** defines activation constraints based on collaborative context and user's membership, and in the case of **Transition**, the implications of the transition relative to the **subject** of the collaboration.

B.1.2 Context and Role-based Control

An implementation of **Collaboration** is responsible for the verification and enforcement of rules concerning initialization, the applying of transitions, and the invoking of command events. An implementation achieves this through features exposed under a **Trigger** interface that describe contextual and role-based constraints that may impose limitations on the possible actions that can be applied by a client relative to the **subject** of the collaboration. These controls may be supplemented through references to a **MembershipKind**, an equivalent of a category of a role relative to a **Membership**.

B.1.3 Applying Compound Transitions

The Collaboration type provides support for specialization of the transition interface called **CompoundTransition**. Compound transitions extend the simple transition model by introducing an alternative destination (used as a reference to the failure transition destination). A more interesting feature of the compound transition is the fact that an **EncounterTemplate** is used to describe the execution semantics of the transition. An implementation of **Collaboration** uses the template to create an instance of **Encounter**, which itself may be another collaboration process. This technique is used extensively in the definition of multilateral negotiation and promissory collaboration models. More importantly, it provides an open mechanism through which arbitrarily complex collaboration patterns can be constructed, encapsulated within transition declaration, and reused within different business processes.

B.1.4 Customization and Extension through Collaboration Models

Collaboration represents the computational view of a collaborative encounter. An instance of collaboration has an association to, and is dependent on, a **CollaborationTemplate**. A **CollaborationTemplate** is composed of a set of customizable building blocks. The building blocks include **State**, **Transition**, and **CompoundTransition**. Each building block can be parameterized by modifying features of the respective types. Association of customized models to collaboration templates enables the creation of libraries of executable process models.

B.2 Usage Scenario

A simple retail model is used to describe the way in which the interfaces that form this specification are applied to the problem of

- evaluation of a collaborative process prior to participation,
- participation to process enabling negotiation resolution of agreement, and
- engagement in the implications of agreement or failure.

B.2.1 Simple Retail Model

The example retail business model is described through an instance of **CollaborationTemplate**. The example collaboration template contains two states "for-sale" and "sold" linked together by a single transition named "purchase." The model expresses as a very simple collaborative process involving an owner of something for-sale, and a potential buyer. In this usage scenario these roles are referred to respectively as supplier and consumer.

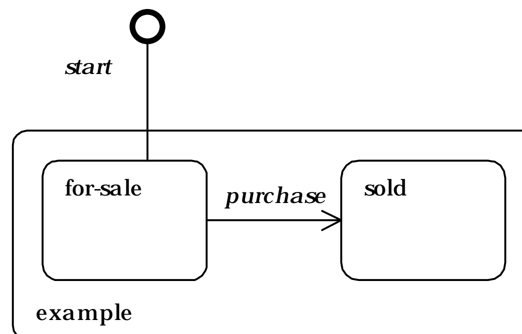

Figure B-2 Example CollaborationTemplate

The scenario as presented above does not include negotiation; however, it does serve as a basic business model example that we will evolve to include both negotiation and implications of negotiated agreement later in this section.

Construction of the Retail Model

A collaborative model is constructed through the population of a collaboration template with instance of states and transitions. The above model is expressed through associating the **label** "example" to the instance of **CollaborationTemplate** and populating the template with two sub-states with the labels "for-sale" and "sold." An instance of **Transition** is added to the template with a reference to the "for-sale" state as **source** and the sold state as **target** of the transition. A second **Transition** is

required to describe the “start” initialization. An initialization transition for this example is flagged as an initializing by the boolean attribute **initialize**, references the “for-sale” state as the **target**, and exposes the **label** “start.”

To describe the allowable collaborative actions, we need to establish the constraints concerning access to these transitions. The simplest approach is to assign the **mode** of **INITIATOR** to the *start* transition and **RESPONDENT** to the *purchase* transition. These constraints qualify the implicit collaborative roles of two participants – one as the initiator and another as a respondent (where respondent is any participant other then the initiator). A second constraint concerns the declaration of restrictions concerning modification of a subject. In our example we will restrict both the start and purchase transitions to be **TRANSITIONAL** (indicating that subject modification is not supported). To complete the state model we need to declare all final states as either a **SUCCESS** or **FAILURE** using the **terminal** attribute on the **State** interface. In our example the “sold” state signifies the successful conclusion of the process.

To restrict the model to two participants, a **MembershipKind** needs to be referenced under the **membership_kind** attribute inherited from the super-type **EncounterTemplate**. In this example, we need a membership kind that established a **quorum** of 2, and a **ceiling** of 2 (corresponding to the initiator and respondent). These values restrict the minimum and maximum number of participants that can join an instance of Collaboration that references this example model.

Operations enabling the construction and population of values under both **CollaborationTemplate** and **MembershipKind** are implementation dependent (i.e., no standard interfaces are defined under this specification – different implementations are required to provide proprietary editors). Interfaces that are exposed include operations needed to navigate a populated model.

Publishing a Collaborative Process

The act of collaboration is separate and distinct from the model of collaboration. An instance of **Collaboration** exposes the operations through which a user may join, interact, and leave the process. Each instance of **Collaboration** references a **CollaborationTemplate** and a **MembershipKind**. **MembershipKind** establishes the rules under which users join and leave collaboration, and together with the **CollaborationTemplate** sets the rules under which participants may interact.

Publishing our example **CollaborationTemplate** can be achieved through publishing an instance of **Collaboration**. **Collaboration** is derived from **Session::Task** and as such may be set in an **open**, **not-running**, **not-started** state and made accessible though inclusion within a **Workspace**.

It is important to note that a supplier or consumer may create a **Collaboration** instance. For example, a supplier could publish instances of **CollaborationTemplate**, enabling each new customer to invoke their own **Collaboration** process. In such a case, a supplier would typically define role-based restrictions that guaranteed the supplier a role in the **Collaboration**.

Navigating a Collaboration

Prior to joining an instance of **Collaboration**, a user having access to the **Collaboration** can navigate exposed relationships. These relationships include the **subject**, membership kind **model**, and the collaboration **template**. The **subject** attribute references an **ActiveResource** that exposes the subject of the collaboration, possibly an XML based product description. The **model** attribute references a **MembershipKind** that qualifies the behavior of the **Collaboration** in terms of membership rules and role. In our example this is limited to the qualification of a **ceiling** and **quorum** required before collaborative operations can be invoked. The **template** attribute references the **CollaborationTemplate** we constructed earlier that exposes the initialization transition, the two sub-states “for-sale” and “sold,” and the “purchase” transition.

Joining a Collaboration

A **Collaboration** is a type of **Membership** and as such exposes the **recruitment_status** attribute. The value of **recruitment_status** is one of the enumerated values **OPEN_MEMBERSHIP**, **CLOSED_MEMBERSHIP**, or **SUSPENDED_MEMBERSHIP**. By setting the recruitment status to **OPEN_MEMBERSHIP** we are advertising the fact that membership to this collaboration is invited. All participants to a **Collaboration** join by invoking the **add_member** operation on the **Collaboration** instance (operation inherited from **Membership**). The participant passes in two arguments, a reference to an **ActiveUser** and a reference to a **MembershipKind** and gets back a reference to **Member** (a role of **ActiveUser**).

An instance of **Collaboration** exposes its readiness for collaborative execution through the attribute **quorum_status**. A quorum status of **QUORUM_PENDING** indicates an insufficient number of participants whereas **QUORUM_REACHED** indicates that the necessary number of participants have joined and that collaborative operations may be invoked. In our example the **quorum** and **ceiling** level are the same, as such, on reaching quorum the **recruitment_status** will change from **OPEN_MEMBERSHIP** to **CLOSED_MEMBERSHIP**.

Initializing the Collaboration

In our example process the initialization transition can be invoked by either of the two participants. In a real example it is more likely that the initialization transition would be associated to a particular **Member** role; however, the example model simply states that whoever initializes the collaboration takes on the implicit role of **INITIATOR**. As initiator, that participant may no longer invoke the *purchase* transition (because *purchase* is restricted to the **RESPONDENT**). This restriction is maintained until a respondent invokes a transition in which case the respondent becomes the initiator and the prior initiator becomes a respondent.

Under the example model there is only one initializing transition (labeled “start”). We assume that the service provider (the user wishing to sell the goods or service) invokes the initialization. Invoking the initialization is achieved by invoking the **apply**

operation on the **Collaboration** instance and passing in the "start" transition as the **transition** argument. As our "start" transition is restricted to **TRANSITIONAL** (exposed under the **Transition** interface control attribute) we cannot change or replace the **subject** of the **Collaboration**.

If the transition control attribute was **PROCESS** instead of **TRANSITIONAL**, we could have supplied the supplementary **semantic** argument of **REPLACE** or **MODIFY**. In the case of a **REPLACE**, a third argument is required corresponding to an **ActiveResource** with which to replace the current **subject**. Alternatively, a semantic argument **MODIFY** together with **ActiveTask** would have declared the task to use to modify the current subject. It is important to note that this specification is independent in respect to the subject of an **Encounter**. It is the responsibility of a client to discover the **subject** type of an **Encounter** and to create an appropriate **ActiveTask** (bound to an editor capable of modifying the **subject** type) through which **subject** modification may be executed.

On invocation of the apply operation, the implementation of **Collaboration** executes the verification of the principal as a register Member of the Collaboration, validates that the applied transition constraints are not being violated, and depending on parameters of the transition invokes the appropriate changes in the **Collaboration** state. In our example, the "start" transition establishes the **active_state** of the collaboration as the state sequence: *example*, *for-sale* (indicating that both the state labeled "example" and the state labeled "for-sale" are active).

Post Initialization Actions

Based on the constraints established under the **CollaborationTemplate**, the supplier is now the **INITIATOR** and our consumer is now **RESPONDENT**. Our example model exposes a single transition that matches its **source** with an active state – the "purchase" transition. The "purchase" transition is restricted to **RESPONDENT**, which eliminates the possibility for the supplier to invoke the transition (because supplier principal is considered the initiator by the implementation and therefore is excluded from the set of possible respondents). At this point our example process is starting to appear somewhat artificial; however, we will continue through the remainder of the process and address more realistic transition models at a later stage.

Our customer invokes the "purchase" transition by passing the transition in under the **apply** operation **transition** argument. The implementation, after verification of compliance with implied collaborative role and transitional constraints, sets the **active state** of the collaboration to *example, sold.*

Process Termination

On establishing *sold* as an active state, the implementation recognizes the terminal value of **SUCCESS** and raises a corresponding **result** event (the **result** event is exposed under the super-type **Encounter**). Prior to completion of the process the **Collaboration** evaluates any implication associations declared under the **template**. Implications are associations that reference other **EncounterTemplate** (the super-type of **CollaborationTemplate**) that have to be invoked relative to the successful or

unsuccessful result of the **Collaboration**. For example, an implication of purchasing could be the instantiation of a payment collection process, or a product warranty process (or both). In our example we have not assigned any implications and as such the **Collaboration** process enters the **closed**, **completed** task state. Note that task state is the state of execution as described by the **Task/Session** specification. This is orthogonal to the **active state** of **Collaboration**.

B.2.2 Introducing a Compound, Negotiable Transition

As indicated above, our example model is too restrictive to realistically represent a commercial retail process. A more realistic example would typically expose several alternative transitions. For example, a transition that enabled rejection of the offered goods and services, and perhaps another transition that enabled modification of a feature of the subject of the **Collaboration** such as quantity or delivery conditions.

To bring our example closer to a realistic model, we are going to replace the simple "purchase" transition with a compound transition. A **CompoundTransition** is a transition that is itself defined by an **EncounterTemplate** (the super-type of **CollaborationTemplate**). In effect, the execution of a compound transition is equivalent to the entry into another **Encounter** where the result of the subsidiary encounter determines the result of the parent transition. The compound transition we are going to use is a bilateral negotiation, expressed under a **CollaborationTemplate**, which will enable the extension of our example to include a negotiable purchase decision.

Purchase as a Bilateral Negotiation

The following illustration depicts the replacement of the "purchase" transition with a compound transition of the same name. Compound transitions have two possible target states, one representing the target-state to establish on success, the other representing the target-state to establish on failure. Success or failure is determined by the **result** status of the execution of the transition as a subsidiary **Encounter** (the super-type of **Collaboration**).

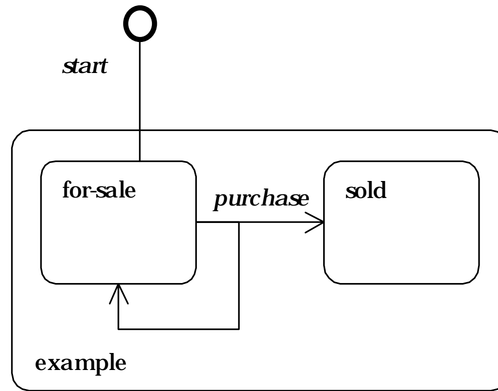

Figure B-3 Example CollaborationTemplate

Our example scenario remains unchanged until our consumer applies the “purchase” transition. As a **CompoundTransition**, “purchase” now exposes two additional attributes of interest, 1) a reference to a **CollaborationTemplate** and 2) a reference to an initializing transition within that **template**. In invocation of the **apply** operation, the implementation of the **Collaboration** establishes a subsidiary **Encounter** (in this example the **Encounter** is another **Collaboration**) with the same **subject**, associated to the **CollaborationTemplate** describing a bilateral (one-on-one) negotiation and referencing the same membership **model**. The second feature of interest is the initialization attribute exposed by the **CompoundTransition**. The initialization attribute references the initialization transition to apply to the subsidiary **Collaboration**, resulting in the establishment of the bilateral negotiation sub-process.

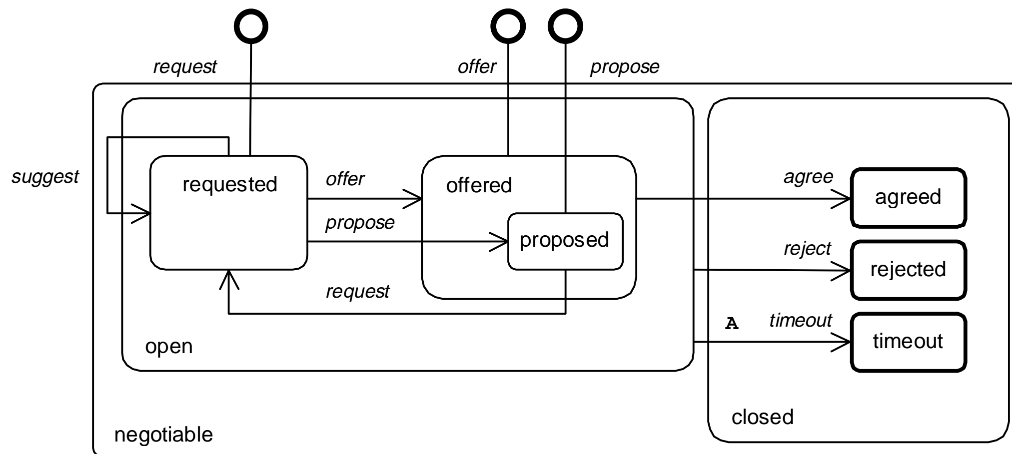

Figure B-4 Bilateral State Transition Model

The above illustration depicts the state transition model of a bilateral negotiation, which following invocation of the purchase transition, is established under *requested*, *offered*, or *proposed* states. Assuming the purchase transition initialization argument referenced the *requested* transition, the active states of the sub-**Collaboration** would be *negotiable*, *open*, and *requested*.

For a detailed description of the semantics of the following state transition model please refer to the bilateral negotiation process model specification under the *SessionFramework* section.

Under the requested state, the respondent may invoke any transition that references an active state as its **source**. For example, *suggest*, *offer*, *propose* (from the *requested* state), or *reject* (from the *open* state). If our respondent invoked *offer*, the available transitions would be restricted to *agree* (from the *offered* state) or *reject* (from the *open* state). If our respondent had chosen the *propose* transition, the transitions available to the correspondent would also include the *request* transition (from the *proposed* state).

It is important to note that the bilateral negotiation state transition model is simply an example of a collaborative process model. This specification does not impose any restriction on the process described within **CollaborationTemplate** beyond the requirement that the semantics of the process are described using the **State**, **Transition**, and associated interfaces documented in the *CommunityFramework* section of this specification.

Through the invocation of transitions in the context of the implied roles of respondent and initiator, our two participants can migrate from a non-agreed to agreed state. During this process, dependent on the constraints imposed by respective transitions, subject modification may be possible (though the declaration of **ActiveResource** as replacement subject or **ActiveTask** as subject modifier). On conclusion of the process, through the establishment of a terminal **SUCCESS** or **FAIL** state, the **Collaboration** process raises a **result** event and terminates. Control is returned to the parent **Collaboration**. Based on the **result** status, the parent **Collaboration** determines the appropriate target-state to establish as the **active state**. Assuming a successful conclusion of the negotiation our active state would be set to “sold.” A failure of the subsidiary negotiation would establish “for-sale” as the active state.

A detailed description of the semantics concerning the interaction between parent and subsidiary process and the relationship and impact of changes to a subject under a subsidiary **Collaboration** are detailed under Section 4.3.1.8, “Applying State Transitions,” on page 4-12.

B.2.3 Introducing Implications

The process of encapsulation of subsidiary processes within compound transitions enables the introduction of complex collaborative models. Another mechanism through which the semantics of collaboration is further enhanced is through the association of a collaboration as the **Implication** of the **success** or **failure** of a prior **Collaboration** (or more correctly, the success or failure of a prior **Encounter)**. To introduce an implication into our example, we need to add the declaration of an implication to our example **CollaborationTemplate** instance. An **Implication** is a

type of **Linkage**. Instances of **Linkage** reference a **source** and a **target** object and are used as arguments to the **bind** and **release** operations exposed by the **ActiveResource** super-type. To associate a payment process as an implication of the success of a purchasing process, we construct an instance of **Success** (a specialization of **Implication**) that references our example template as the **source** of the linkage (success implication source) and the payment process template as the **target** of the linkage. The object model allows for the association of many implications relative to a given source. For example, the implications of a successful purchasing process may also include the establishment of a delivery process, which itself may have a success implication of a maintenance contract. Association of the source example template and a target payment process is achieved by invoking the **bind** operation on the **source** and **target**.

B.2.4 Comparing the Example to the Promissory Encounter Model

The promissory encounter process model is simply an enhanced version of the example process model presented here (see Figure B-5).

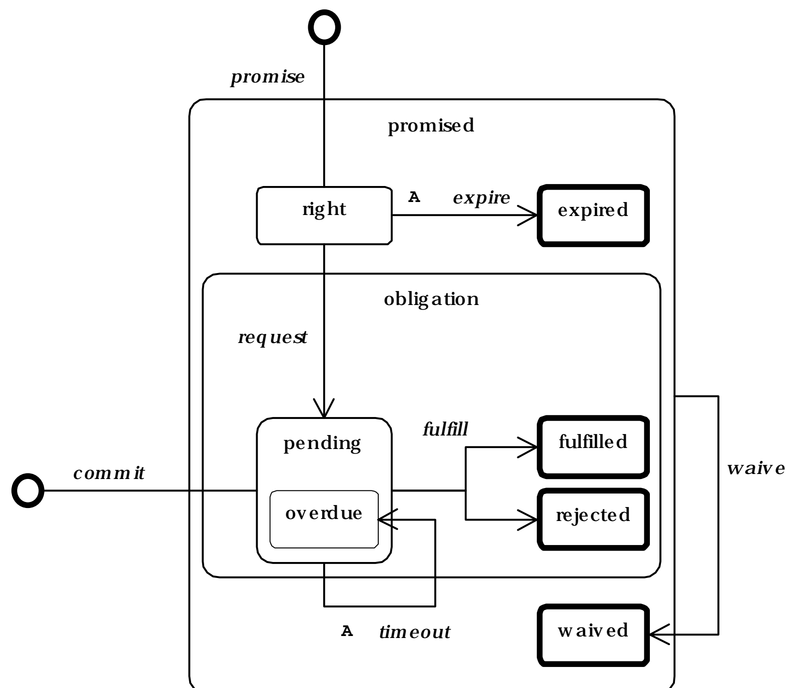

Figure B-5 Promissory State Transition Model

Instead of "for-sale" and "sold" we have the states "*promised, obligation, pending*" and "*promised, obligation, fulfilled*" and a compound transition "*fulfill*" that links the two states such that the achievement of fulfillment is expressed as a bilateral negotiation between the promise holder and the promise provider. A second occurrence

of a compound transition referencing the bilateral negotiation model is the *waive* transition. A successful transition under *waive* results in the establishment of the "*promised, waived*" state; whereas, failure of the *waive* transition results in the continuation of the collaboration without a change in active state.

For additional information concerning the promissory encounter model, refer to Section 2.2, "ActiveResource and Associative Interfaces," on page 2-5.

Index

A
AbstractTemplate 2-21
Access control based on Membership 4-20
Activation Semantics 4-20
Active State 4-11
ActiveResource 2-5
ActiveTask 2-11
ActiveUser 2-18
ActiveWorkspace 2-15
Agency 3-18
Agree 4-33
Amendment 4-36
Apply Exceptions 4-14
Applying State Transitions 4-12

B
Batch and Interactive Modes 2-13

C
Call 4-37
Collaboration 4-8
CollaborationFramework 4-1
CollaborationTemplate 4-10, 4-15
Collaborative Context 4-19
Command 4-21
Community 3-17
Community and Derived Interfaces 3-16
Composition 2-11
CompoundTransition 4-22
Containment 2-17
CORBA
  documentation set 1-3
Count 4-38

D
Delegation 2-10
Desktop 2-16
DocumentResource 5-3
DOM Level 1 IDL 5-7
DomFramework Wrapper Interfaces 5-1

E
Encounter 4-3
EncounterTemplate 4-6
Engagement 4-24
EngagementManifest 4-26
EngagementTemplate 4-24
Execution Modes 4-19
Expire 4-42
Extensions 5-3

F
Fulfill 4-41

I
Implication 4-6
Implication Semantics 4-5
Initialization 4-5, 4-31, 4-40
Initialization of a Collaboration 4-12
Invoking Command Events 4-15

J
Jurisdiction 2-20

L
LegalEntity 2-19
Linkage 2-8
Linkage Dependencies 2-6
Linkage Types 2-3
Listing Kind Attributed to a Member 3-6

M
Member 3-5
Member Addition 3-9
Member Removal 3-10
Membership 3-6
Membership Composition 3-13
Membership Disclosure Operations 3-12
Membership Semantics 3-8
Membership, Associative and Qualifying Interfaces 3-3
MembershipKind 3-14
Motion 4-36
multilateral encounter 4-34

O
Object Management Group 1-1
  address of 1-4
Offer 4-33

P
Process 2-15
Promissory Encounter 4-40
Propose 4-32

Q
Quorum Status 3-11

R
Recruitment Status 3-10
Registering a Vote 4-29
Reject 4-33
Request 4-32, 4-41
Resource Usage 2-13

S
Second 4-36
SessionFramework 2-1
State 4-17
State Composition 4-18
States 4-34
Structural Operations 3-12
Structured Events 2-7
Subject Modification Constraints 4-22
Subsidiary Collaboration Processes 4-11
Suggest 4-32

T
Task/Session interfaces 2-1
Timeout 4-33, 4-38, 4-42
Timeout behavior 4-11
Transition 4-21
Transitions 4-32, 4-36, 4-41
Trigger 4-18

Index

Trigger Lifetime 4-20

U
Usage 2-14

V
Vote 4-37
VoteManifest 4-29
VoteTemplate 4-28
Voting 4-27

W
Waive 4-41
Withdraw 4-38