Concurrency Service Specification

Version 1.0
New Edition: April 2000

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

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.

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. CORBA 1.1 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). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *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 Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.
- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.

- *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

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*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Service Design Principles

Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as “building blocks.”

Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer, PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service.
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms.

Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

Interface Style Consistency

Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

Explicit Versus Implicit Operations

Operations are always explicit rather than implied (e.g., by a flag passed as a parameter value to some “umbrella” operation). In other words, there is always a distinct operation corresponding to each distinct function of a service.

Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by “normal” clients.

Acknowledgments

The following companies submitted and/or supported parts of the *Concurrency Service* specification:

- Digital Equipment Corporation
- Electronic Data Systems
- Emeraude
- Groupe Bull
- ICL plc
- Thomson CSF - Syseca

Service Description 1

Contents

This chapter contains the following topics.

1.1 Overview

The purpose of the Concurrency Control Service is to mediate concurrent access to an object such that the consistency of the object is not compromised when accessed by concurrently executing computations.

The Concurrency Control Service consists of multiple interfaces that support both transactional and non-transactional modes of operation. The user of the Concurrency Control Service can choose to acquire locks in one of two ways:

- On behalf of a transaction (transactional mode.) The Transaction Service drives the release of locks as the transaction commits or aborts.
- By acquiring locks on behalf of the current thread (that must be executing outside the scope of a transaction). In this non-transactional mode, the responsibility for dropping locks at the appropriate time lies with the user of the Concurrency Control Service.

The Concurrency Control Service ensures that transactional and non-transactional clients are serialized. Hence a non-transactional client that attempts to acquire a lock (in a conflicting mode) on an object that is locked by a transactional client will block until the transactional client drops the lock.

1.2 Basic Concepts of Concurrency Control

1.2.1 Clients and Resources

The Concurrency Control Service enables multiple *clients* to coordinate their access to shared *resources*. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

The Concurrency Control Service does not define what a resource is. It is up to the clients of the Concurrency Control Service to define resources and to properly identify potentially conflicting uses of those resources. In a typical use, an object would be a resource, and the object implementation would use the concurrency control service to coordinate concurrent access to the object by multiple clients.

1.2.2 Transactions as Clients

The Concurrency Control Service differentiates between two types of client: a transactional client and a non-transactional client. Conflicting access by clients of different types is managed by the Concurrency Control Service, thereby ensuring that clients always see the resource in a consistent state.

The Concurrency Control Service does not define what a transaction is. Transactions are defined by the Transaction Service. The Concurrency Control Service is designed to be used with the Transaction Service to coordinate the activities of concurrent transactions.

The Transaction Service supports two modes of operation: implicit and explicit. When operating in the implicit mode, a transaction is implicitly associated with the current thread of control. When executing in the explicit mode, a transaction is specified explicitly by the reference to the coordinator that manages the current transaction. To simplify the model of locking supported by the Concurrency Control Service when a transactional client is operating in the implicit transaction mode, transactional clients are limited to a single thread per transaction (nested transactions can be used when parallelism is necessary) and that thread can be executing on behalf of at most one transaction at a time.

1.2.3 Locks

The Concurrency Control service coordinates concurrent use of a resource using locks. A lock represents the ability of a specific client to access a specific resource in a particular way. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the activities of those clients might conflict. To achieve coordination, a client must obtain an appropriate lock before accessing a shared resource.

1.2.4 Lock Modes

The Concurrency Control Service defines several *lock modes*, which correspond to different categories of access. Having a variety of lock modes allows more flexible conflict resolution. For example, providing different modes for reading and writing allows a resource to support multiple concurrent clients that are only reading the data of the resource. The Concurrency Control Service also defines *intention locks* that support locking at multiple levels of granularity.

1.2.5 Lock Granularity

The Concurrency Control Service does not define the granularity of the resources that are locked. It defines a *lock set*, which is a collection of locks associated with a single resource. It is up to clients of the Concurrency Control Service to associate a lock set with each resource. Typically, if an object is a resource, the object would internally create and retain a lock set. However, the mapping between objects and resources (and lock sets) is up to the object implementation; the mapping could be one to one, but it could also be one to many, many to many, or many to one.

1.2.6 Conflict Resolution

A client obtains a lock on a resource using the Concurrency Control Service. The service will grant a lock to a client only if no other client holds a lock on the resource that would conflict with the intended access to the resource. The decision to grant a lock depends upon the modes of the locks held or requested. For example, a read lock conflicts with a write lock. If a write lock is held on a resource by one client, a read lock will not be granted to another client.

1.2.7 Conflict Resolution for Transactions

The decision to grant a lock also depends upon the relationships among the transactions that hold or request a lock. In particular, if the transactions are related by nesting (nested transactions), a lock may be granted that would otherwise be denied.

1.2.8 Lock Duration

Typically, a transaction will retain all of its locks until the transaction is completed (either committed or aborted). This policy supports serializability of transactional operations. Using the two phase commit protocol, locks held by a transaction are automatically dropped when the transaction completes.

There are also situations where levels of isolation that are weaker than serializability are acceptable, such as when an application does not want other applications to change an object while reading it and does not refer to the object again within the transaction. In these circumstances, it is acceptable to release locks before the containing transaction completes, hence the duration will be shorter than the containing transaction.

To manage the release of the locks held by a transaction, the Concurrency Control service defines a lock coordinator. Lock sets that are related (for example, by being created by a resource manager for resources of the same type) and that should drop their locks together when a transaction commits or aborts may share a lock coordinator. It is up to clients of the concurrency control service to associate lock sets together and to release the locks when a transaction commits or aborts.

1.3 Locking Model

This section covers a number of important issues that relate to the locking model supported by the Concurrency Control Service. For a complete discussion of these issues the reader is directed to one of the standard texts on the subject[1].

The Lock Modes section applies to clients that operate in both transactional and non-transaction modes. The Multiple Possession Semantics, Two-Phase Transactional Locking, and Nested Transaction sections are relevant only to clients that operate in transactional mode.

1.3.1 Lock Modes

1.3.1.1 Read, Write, and Upgrade Locks

The Concurrency Control service defines *read* (R) and *write* (W) lock modes that support the conventional multiple readers, one writer policy. Read locks conflict with write locks, and write locks conflict with other write locks.

In addition, the Concurrency Control service defines an *upgrade* (U) mode. An upgrade mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock that occurs when two or more clients attempt to read and then update the same resource. If more than one client holds a read lock on the resource, a deadlock will occur as soon as one of the clients requests a write lock on the resource. If each client requests a single upgrade lock followed by a write lock, this deadlock will not occur.

1. See *Concurrency Control and Recovery in Database Systems* by P.A. Bernstein, V. Hadzilacos, and N. Goodman, or *Transaction Processing: Concepts and Techniques* by J.N. Gray and A. Reuter.

1.3.1.2 *Intention Read and Intention Write Locks*

The granularity of the resources locked by an application determines the concurrency within the application. Coarse granularity locks incur low overhead (since there are fewer locks to manage) but reduce concurrency since conflicts are more likely to occur. Fine granularity locks improve concurrency but result in a higher locking overhead since more locks are requested. Selecting a suitable lock granularity is a balance between the lock overhead and the degree of concurrency required. Using the Concurrency Control service, an application can be developed to use coarse or fine granularity locks by defining the associated resources appropriately.

In addition, the Concurrency Control service supports variable granularity locking using two additional lock modes, *intention read* (IR) and *intention write* (IW). These additional lock modes are used to exploit the natural hierarchical relationship between locks of different granularity.

For example, consider the hierarchical relationship inherent in a database: a database consists of a collection of files, with each file holding multiple records. To access a record, a coarse grain lock may be set on the database, but at the cost of restricting other clients from accessing the database. Clearly, this level of locking is unsuitable. However, only setting a lock on the record is also inappropriate, because another client might set a lock on the file holding the record and delete or modify the file.

Using variable granularity locking, a client first obtains intention locks on the ancestor(s) of the required resource. To read a record in the database, for example, the client obtains an intention read lock (IR) on the database and the file (in this order) before obtaining the read lock (R) on the record. Intention read locks (IR) conflict with write locks (W), and intention write locks (IW) conflict with read (R) and write (W) locks.

1.3.1.3 *Lock Mode Compatibility*

Table 1-1 Lock Compatibility

Granted Mode	**Requested Mode**				
	IR	**R**	**U**	**IW**	**W**
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

Table 1-1 defines the compatibility between the various locking modes (the symbol * is used to indicate when locks conflict). When a client requests a lock on a resource that cannot be granted because another client holds a lock on the resource in a conflicting mode, the client must wait until the holding client releases its lock. The Concurrency Control Service enforces a queueing policy such that all clients waiting for a new lock are serviced in a first in, first out order, and subsequent requests are blocked by the first request waiting to be granted the lock, unless the requesting client is a transaction that is a member of the same transaction family as an existing holder of the lock.

1.3.2 Multiple Possession Semantics

The Concurrency Control Service interface supports a locking model called multiple possession semantics. In this model, a client can hold multiple locks on the same resource simultaneously. The locks can be of different modes. In addition, a client can hold multiple locks of the same mode on the same resource; effectively, a count is kept of the number of locks of a given mode that have been granted to the client. When a client holds locks on a resource in more than one mode, other clients will not be granted a lock on the resource unless the requested lock mode is compatible with all of the modes of the existing locks.

In contrast, using the conventional locking model,[2] when a client holding a lock on a resource requests a lock on the same resource in a stronger mode, the existing lock is promoted from the weaker mode to the stronger mode (once the stronger lock can be granted without causing a conflict). Since lock modes form only a partial order, there will not always be a stronger mode; in cases where neither mode is stronger, the lock will be promoted to the weakest mode that is at least as strong as either of the two modes.

1.4 Two-Phase Transactional Locking

The Concurrency Control Service provides primitives to support transaction-duration locking. Transaction duration locking is a special case of strict two-phase locking. In the first phase (the growing phase), a transaction obtains locks that are kept until the second phase (the shrinking phase), at which point they are released. A transaction must not release locks during the first phase, and must not obtain new locks during the second phase, otherwise concurrent computations may be able to view intermediate results of the transaction.

Two-phase locking is sufficient to guarantee serializability, hence this technique is used by transactions. During the normal execution of a transaction, no locks will be automatically dropped before the end of the transaction. When the transaction completes, the Concurrency Control Service must be informed so that the locks the transaction holds may be released. While releasing locks, no new locks may be obtained by the transaction.

2. See *Notes On Data Base Operating Systems* in *Operating Systems: An Advanced Course* (ed. Bayer, Graham, and Seegmuller) by J.N. Gray for further information.

When a transaction holds a lock that is no longer needed to ensure the transaction's serializability, or if a weaker level of isolation is acceptable, it is permissible to release the lock. The Concurrency Control Service therefore provides an operation that releases individual locks. This operation should be used with caution to ensure that the isolation level is appropriate for the application.

1.5 Nested Transactions

Lock conflicts within a transaction family are treated somewhat differently than conflicts between unrelated transactions. The underlying principle is the same for both: transactions must not be able to observe the effects of other transactions that might later abort. Unrelated transactions can abort independently; therefore, one transaction must not be permitted to acquire a lock that conflicts with a lock on the same resource held by an unrelated transaction.

Nesting imposes abort dependencies among related transactions. A parent transaction cannot abort without causing all of its children to abort. A child transaction that ends successfully cannot abort without causing its parent to abort. A transaction that cannot abort without causing another related transaction to abort (according to these guidelines and logical deductions) is said to be committed relative to that other transaction.

These dependencies make it possible to relax the rule that two transactions cannot acquire locks of conflicting modes on the same resource, without breaking the underlying principle. No partial effects can be observed and committed if all transactions that have done work cannot abort without the observer being aborted. This property translates into a simple rule for nested locking: if all transactions holding locks on a resource are committed with respect to a transaction trying to acquire a lock on the resource, no conflict exists.

The multiple possession model (see previous section) facilitates the use of locks with nested transactions. In this model, multiple related transactions may hold locks of conflicting modes on a resource at the same time. When a nested transaction requests a lock, it is granted if all of the transactions holding locks on the resource are committed relative to the requestor. Both the requestor and previous holders are then considered to hold locks on the resource.

A child transaction can acquire a lock on a resource locked by its parent and then drop that lock without causing its parent to lose its lock. A transaction cannot drop a lock that it did not acquire itself. The lock possession semantics also require that each transaction acquire locks on its own behalf. It is improper to take locks on behalf of another transaction or to depend on locks held by other transactions.

Other approaches to nested transactions[3] treat a resource as being locked by a single transaction at a time. When a nested transaction requests a lock on a resource that is already locked by an ancestor transaction, the nested transaction becomes the new

3. See *Nested Transactions: An Approach To Reliable Distributed Computing* by J.E.B. Moss for further information.

owner of the lock. When a nested transaction commits, ownership of all of its locks is transferred to its parent. When a nested transaction aborts, ownership of its locks reverts to the previous owners. The Concurrency Control service performs these lock transfers automatically. The multiple possession semantics model is functionally equivalent to this model, but it supports simpler interfaces.

Modules and Interfaces 2

Contents

This chapter contains the following topics.

2.1 *CosConcurrencyControl Module*

The Concurrency Control Service is defined by the **CosConcurrencyControl** module, which provides interfaces that support both transactional and non-transactional modes of operation. This section defines the interfaces and describes the operations they support.

- The interfaces provide two modes of operation that correspond to those supported by the Transaction Service; in both modes, locks are identified by the lock set they are associated with and the mode of the lock.
- A client of the Concurrency Control Service may operate in an implicit mode such that locks are acquired on behalf of the current transaction (for transactional clients) or current thread (for non-transactional clients).
- For transactional clients, a second alternative is possible that involves the client identifying the transaction by means of a reference to the transaction's coordinator object (the explicit mode of operation).

Locks are acquired on lock sets. Two sets of operations are provided by the **LockSetFactory** interface to create lock sets, one creates a lock set that can be used by clients operating in the implicit mode (the **LockSet** interface), the other creates a lock set for explicit mode transactional clients (the **TransactionalLockSet** interface). In addition, the **LockCoordinator** interface is provided to allow a client to release all locks held by a specific transaction.

The following sections define the types and exceptions common to both types of interface, the interfaces themselves, and describes the responsibilities of a user for managing transaction-duration locks.

OMG IDL for the **CosConcurrencyControl** module.

```
#include <CosTransactions.idl>
module CosConcurrencyControl {

       enum lock_mode {
             read,
            write,
            upgrade,
            intention_read,
            intention_write
   };

   exception LockNotHeld{};

   interface LockCoordinator
   {
      void drop_locks();
   };

   interface LockSet
      {
      void lock(in lock_mode mode);
      boolean try_lock(in lock_mode mode);

      void unlock(in lock_mode mode)
         raises(LockNotHeld);
      void change_mode(in lock_mode held_mode,
                   in lock_mode new_mode)
        raises(LockNotHeld);
      LockCoordinator get_coordinator(
         in CosTransactions::Coordinator which);
   };

   interface TransactionalLockSet
   {
      void lock(in CosTransactions::Coordinator current,
             in lock_mode mode);
      boolean try_lock(in CosTransactions::Coordinator current,
                   in lock_mode mode);
        void unlock(in CosTransactions::Coordinator current,
                in lock_mode mode)
          raises(LockNotHeld);
        void change_mode(in CosTransactions::Coordinator current,
                     in lock_mode held_mode,
                     in lock_mode new_mode)
          raises(LockNotHeld);
        LockCoordinator get_coordinator(
            in CosTransactions::Coordinator which);
    };

    interface LockSetFactory
    {
        LockSet create();
        LockSet create_related(in LockSet which);
        TransactionalLockSet create_transactional();
        TransactionalLockSet create_transactional_related(in
            TransactionalLockSet which);
    };
};
```

2.1.1 Types and Exceptions

The types and exceptions described in this section apply to both the **Lockset** and **TransactionalLockset** interfaces.

```
module CosConcurrencyControl {
    enum lock_mode {
        read,
        write,
        upgrade,
        intention_read,
        intention_write
    };

exception LockNotHeld{};
```

lock_mode

The **lock_mode** type represents the types of lock that can be acquired on a resource.

LockNotHeld

The LockNotHeld exception is raised when an operation to unlock or change the mode of a lock is called and the specified lock is not held.

2.2 LockCoordinator Interface

The **LockCoordinator** interface enables a transaction service to drop all locks held by a transaction. The **LockSet** and **TransactionalLockSet** interfaces create instances of the **LockCoordinator** for each transaction. The **LockCoordinator** interface provides a single operation:

```
interface LockCoordinator {
   void drop_locks();
};
```

drop_locks

Releases all locks held by the transaction. This call is designed to be used by transactional clients when a transaction commits or aborts. For nested transactions, this operation must be called when the nested transaction aborts, but the call need only be made once for a transaction family when that family commits (recall that nested transaction commits are handled implicitly by the Concurrency Control service).

2.3 LockSet Interface

For clients operating in the implicit mode, locks are acquired and released on lock sets which are defined by means of the **LockSet** interface. The **LockSet** interface only provides operations to acquire and release locks on behalf of the calling thread or transaction. The interface does not provide support for transactional clients that use the explicit Transaction Service interfaces.

```
interface LockSet {
   void lock(in lock_mode mode);

   boolean try_lock(in lock_mode mode);

   void unlock(in lock_mode mode)
.     raises(LockNotHeld);

   void change_mode(in lock_mode held_mode,
               in lock_mode new_mode)
      raises(LockNotHeld);

   LockCoordinator get_coordinator(in
      CosTransactions::Coordinator which);
};
```

When calls to acquire or release locks are made outside the scope of a transaction then it is assumed that the client is operating in the *non-transactional* mode (the concurrency control implementation must use the appropriate Transaction Service operation to determine whether the current thread is executing on behalf of a transaction).

lock

Acquires a lock on the specified lock set in the specified mode. If a lock is held on the same lock set in an incompatible mode by another client then the operation will block the calling thread of control until the lock is acquired. If a call that is on behalf of a transactional client is blocked and the transaction is aborted then the call will return with the Transactions::TransactionRolledBack exception.

try_lock

Attempts to acquire a lock on the specified lock set. If the lock is already held in an incompatible mode by another client then the operation returns a FALSE result to indicate that the lock could not be acquired.

unlock

Drops a single lock on the specified lock set in the specified mode (recall that a lock can be held multiple times in the same mode). Calls to drop a lock that is not held result in the LockNotHeld exception being raised

change_mode

Changes the mode of a single lock (recall that multiple locks may be held on the same lock set). If the new mode conflicts with an existing mode held by an unrelated client, then the **change_mode** operation blocks the calling thread of control until the new mode can be granted. Like the lock call, if the client is a transaction and it aborts while the thread of control if blocked then the **Transactions::TransactionRolledBack** exception will be raised. Similarly, when a call is made to change the mode of a lock, but the lock is not held in the specified mode, the LockNotHeld exception will be raised.

get_coordinator

Returns the lock coordinator associated with the specified transaction.

2.4 TransactionalLockSet Interface

The **TransactionalLockSet** interface provides operations to acquire and release locks on a lock set on behalf of a specific transaction. The operations that make up the **TransactionalLockSet** interface are:

```
interface TransactionalLockSet {
   void lock(in CosTransactions::Coordinator which,
           in lock_mode mode);

   boolean try_lock(in CosTransactions::Coordinator which,
                in lock_mode mode);

   void unlock(in CosTransactions::Coordinator which,
            in lock_mode mode)
      raises(LockNotHeld);
```

```
  void change_mode(in CosTransactions::Coordinator which,
            in lock_mode held_mode,
            in lock_mode new_mode)
    raises(LockNotHeld);

  LockCoordinator get_coordinator(in
    CosTransactions::Coordinator which);
};
```

The operations provided by the **TransactionalLockSet** interface operate in an identical manner to the equivalent operations provided by the **LockSet** interface. The interfaces differ in that for the **TransactionalLockSet** interface the identity of the transaction is passed explicitly as a reference to the coordinator for the transaction instead of implicitly through an association with the calling thread.

2.5 LockSetFactory Interface

Lock sets are created using the **LockSetFactory** interface.

```
interface LockSetFactory {
  LockSet create();
  LockSet create_related(in LockSet which);

  TransactionalLockSet create_transactional();
  TransactionalLockSet
    create_transactional_related(in
      TransactionalLockSet which);
};
```

This interface provides two sets of operations that return new **LockSet** and **TransactionalLockSet** instances.

create

Creates a new lock set and lock coordinator.

create_related

Creates a new lock set that is related to an existing lock set. Related lock sets drop their locks together.

create_transactional

Creates a new transactional lock set and lock coordinator for explicit mode transactional clients.

create_transactional_related

Creates a new transactional lock set that is related to an existing lock set. Related lock sets drop their locks together.