Relationship Service Specification 9

9.1 Service Description

Distributed objects are frequently used to model entities in the real world. As such, distributed objects do not exist in isolation. They are related to other objects.

Consider some examples of real world entities and relationships:

- A person ***owns*** cars; a car is ***owned by*** one or more persons.
- A company ***employs*** one or more persons; a person is ***employed by*** one or more companies.
- A document ***contains*** figures; a figure is ***contained in*** a document.
- A document ***references*** a book; a book is ***referenced by*** one or more documents.
- A person ***checks out*** books from libraries. A library ***checks out*** books to people. A book is ***checked out*** by a person from a library.

These examples demonstrate several relationships:

- Ownership relationships between people and cars
- Employment relationships between companies and people
- Containment relationships between documents and figures
- Reference relationships between books and documents
- Check out relationships between people, books and libraries.

Such relationships can be characterized along a number of dimensions:

Type

Related entities and the relationships themselves are typed. In the examples, ***employment*** is an relationship defined between ***people*** and ***companies.*** The type of the relationship constrains the types of entities in the relationship; a company cannot employ a monkey since a monkey is not a person. Furthermore, employment is distinct from other relationships between people and companies.

The roles of entities in relationships

A relationship is defined by a set of roles that entities have. In an employment relationship, a company plays an ***employer*** role and a person plays an ***employee*** role.

A single entity can have different roles in distinct relationships. Notice that a person can play the owner role in an ownership relationship and the employee role in an employment relationship.

Degree

Degree refers to the number of required roles in a relationship. The check out relationship is a ternary relationship; it has three roles: the borrower role, the lender role and the material role. A person plays the borrower role, a library plays the lender role and a book plays the material role. Ownership, employment, containment and reference, on the other hand, are of degree 2, or binary relationships.

Cardinality

For each role in a relationship type, the maximum cardinality specifies the maximum number of relationships that may involve that role.

The containment relationship is a many-to-one relationship; a document contains many figures; a figure is contained in exactly one document. A many-to-many relationship is between two sets of entities. The ownership example is a many-to-many relationship; a person can own multiple cars; a car can have multiple owners. The check out relationship is a many-to-one-to-many relationship. A person can check out many books from many libraries. A book is checked out by one person from one library and a library can loan many books to many people.

Relationship Semantics

Relationships often have relationship-specific semantics; that is they define operations and attributes. For example, ***job title*** is an attribute of the employment relationship, while it is not an attribute of an ownership relationship. Similarly, ***due date*** is an attribute of the check out relationship.

For more discussion on object-oriented modeling and design with relationships, see [2].

9.1.1 Key Features of the Relationship Service

- The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: *relationships* and *roles*. A role *represents* a CORBA object in an relationship. A relationship is created by passing a set of roles to a relationship factory.
- Relationships of arbitrary degree can be defined.
- Type and cardinality constraints can be expressed and checked. Exceptions are raised when cardinality and type constraints are violated. The Relationship Service does not define a new type system. Instead, the IDL type system is used to represent relationship and role types. This allows the service to leverage CORBA solutions for type federation.

- The *Relationship* interface can be extended to add relationship specific attributes and operations. Similarly, the *Role* interface can be extended to add role specific attributes and operations.
- The Relationship Service defines three levels of service: base, graph, and specific.
- The base level defines relationships and roles.
- When objects are related, they form graphs of related objects. The graph level extends the base level service with nodes and traversal objects. Traversal objects iterate through the edges of a graph. Traversals are useful in implementing compound operations on graphs, among other things.
- Specific relationships are defined by the third level.

4. A conforming Relationship Service implementation must implement level 1 or levels 1 and 2 or levels 1, 2 and 3.
   - Appendix 6A, which contains an addendum to the Life Cycle Service, defines operations to copy, move, and remove graphs of related objects.
   - The Relationship Service requires a notion of object identify. As such, it defines a simple, efficient mechanism for supporting object identity in a heterogeneous, CORBA-based environment. We believe the mechanism to be of general utility for other services.
   - Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references; role objects can be collocated with their objects and need not depend on a centralized repository of relationship information. As such, navigating a relationship can be a local operation.
   - The Relationship Service allows so-called immutable objects to be related. There are no required interfaces that objects being related must support. As such, objects whose state and implementation were defined prior to the definition of the Relationship Service can be related objects.
   - The Relationship Service allows graphs of related objects to be traversed without activating related objects.
   - The Relationship Service is extensible. Programmers can define additional relationships.

9.1.2 The Relationship Service vs. CORBA Object References

CORBA: Common Object Request Broker Architecture and Specification defines object references that clients use to issue requests on objects. Object references can be stored persistently. When is it appropriate to use object references and when is it appropriate to use the Relationship Service?

The Relationship Service is appropriate to use when an application needs any of the following capabilities that are not available with CORBA object references:

Relationships that Are Multidirectional

When objects are related using the Relationship Service, the relationship can be navigated from any role to any other role. The service maintains the relationship between related objects. CORBA object references, on the other hand, are unidirectional. Objects that posses CORBA object references to each other can only do so in an ad hoc fashion; there is no way to maintain and manipulate the relationship between the objects.

Relationships that Allow Third Party Manipulation

Since roles and relationships are themselves CORBA objects, they can be exported to third parties. This allows third parties to manipulate the relationship. For example a third party could create, destroy or navigate the relationship. Third parties cannot manipulate object references.

Traversals that Are Supported for Graphs of Related Objects

When objects are related using the Relationship Service, they form graphs of related objects. Interfaces are defined by the Relationship Service to support traversing the graph.

Relationships and Roles that Can Be Extended with Attributes and Behavior

Relationships have relationship-specific semantics. For example, the employment relationship has a job title attribute. Since relationships and roles are objects with well-defined OMG IDL interfaces, they can be extended through OMG IDL inheritance to add such relationship-specific attributes and operations.

9.1.3 Resolution of Technical Issues

Modeling and Relationship Semantics

An application designer models a problem as a set of objects and the relationships between those objects. Using OMG IDL, the application designer directly represents the objects of the model. Using the Relationship Service, the application designer directly represents the roles and relationships of the model.

The *Relationship* and *Role* interfaces can be extended using OMG IDL inheritance to add relationship and role specific attributes and operations. For example, a designer might define the employment relationship to have an operation returning a job title.

Managing Relationships

The *RelationshipFactory* interface defines an operation to create a relationship, given a set of roles. The *Role* and *Relationship* interfaces define operations to delete and navigate relationships between objects.

Constraining Relationships

Type, cardinality and degree constraints on relationships are expressed in the interfaces.

The `RoleFactory::create_role` operation can raise a `RelatedObjectTypeError` exception. This allows implementations of the *Role* interface to place further constraints on the type of the related objects. For example, an *EmployedByRole* can ensure related objects are people. An attempt to have it represent a monkey would raise a `RelatedObjectTypeError` exception.

Similarly, the `RelationshipFactory::create` operation can raise a `RoleTypeError` exception. This allows implementations of the *Relationship* interface to put constraints on the type of the roles. For example an Employment relationship can ensure there is an *EmployerRole* and an *EmployeeRole*.

The `RelationshipFactory::create` operation can also raise a `DegreeError` exception. This ensures that there are the correct number of roles.

Maximum cardinality constraints are enforced by the role objects themselves. A role can raise a `MaxCardinalityExceeded` exception and refuse to participate in a relationship if its maximum cardinality would be exceeded. Roles define an operation to ask if their minimum cardinality constraint is being met.

Referential Integrity

If the Relationship Service is used in an environment supporting transactions, strict referential integrity is achieved. That is, if an related object refers to another (via a relationship), then the other related object will also refer to it. Without transactions, strict referential integrity cannot be achieved since a failure during execution of the relationship construction protocol could cause a dangling reference.

Relationships and Roles as First Class Objects

Our design defines both relationships and roles as first class objects. This is extremely important because it encapsulates and abstracts the state to represent the relationship, allows third party manipulation of the relationship and allows the roles and relationships themselves to support operations and attributes.

Different Models for Navigating and Constructing Relationships

The Relationship Service defines interfaces for constructing and navigating relationships component-by-component. These building block operations can be used by a higher-level service, such as a query service.

Efficiency Considerations

Our design has several features that allow for highly optimized implementations. Performance optimizations are achieved by clustering and/or *c*aching of connection information.

Clients can cluster related objects and their roles by their selection of factories.

Our design defines the containment relationship logically. It does not imply physical clustering of state or execution, However, it serves as a good hint to implementations for clustering. An environment can choose to cluster containers and contained objects.

The `get_other_related_object` operation can be implemented to *cache* remote related objects. The cached information is immutable; once a relationship is established, the roles and related objects will not change.

9.2 Service Structure

This section provides information about the levels of service; the specification is organized around these levels. It also describes the hierarchy of Relationship Service interfaces and explains the main purpose of each interface.

9.2.1 Levels of Service

The Relationship Service defines three levels of service: base relationships, graphs of related objects, and specific relationships. The specification is organized around these levels.

Level One: Base Relationships

The *Relationship* and *Role* interfaces define the base Relationship Service. Figure 9-1 illustrates two instances of the containment relationship. The document plays the container role; the figure and the logo play the containee role.

The diamond is an object supporting the *Relationship* interface. The small circles are objects supporting the *Role* interface.

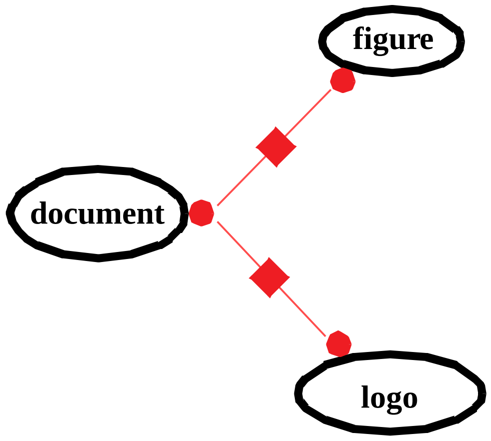

Figure 9-1 Base relationships.

Roles represent objects in relationships. Roles have a maximum cardinality. As illustrated, the container role can be involved in many instances of a relationship. The containee roles can only be involved in a single instance of a relationship.

Figure 9-2 illustrates the navigation functionality of relationships; for example the arrow between a role and another role indicates it is possible to navigate from one role to another. The arrow does not, however, indicate that the object reference to the other role is necessarily stored by the role.

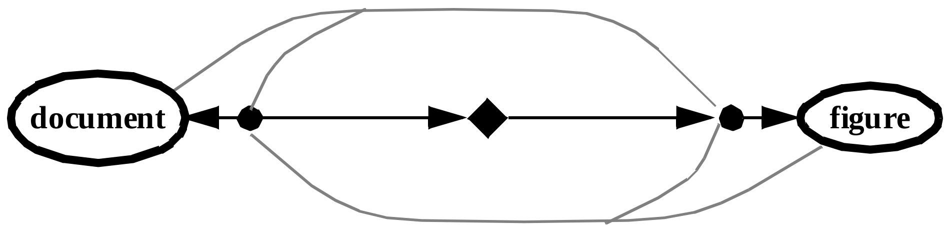

Figure 9-2 Navigation functionality of base relationships

Table 9-1 lists the interfaces to support relationships and roles. Section 9 specifies the interfaces in detail.

Level Two: Graphs of Related Objects

Distributed objects do not exist in isolation. They are connected together. Objects connected together form graphs of related objects. The Relationship Service defines the *Traversal* interface. The *Traversal* interface defines an operation to traverse a graph. The traversal object cooperates with extended roles supporting the *CosGraphs::Role* interface and objects supporting the *Node* interface.

Figure 9-3 illustrates a graph of related objects. The folder, the figure, the logo and the book all support the *Node* interface. The small circles are roles supporting the *CosGraphs::Role* interface.

.

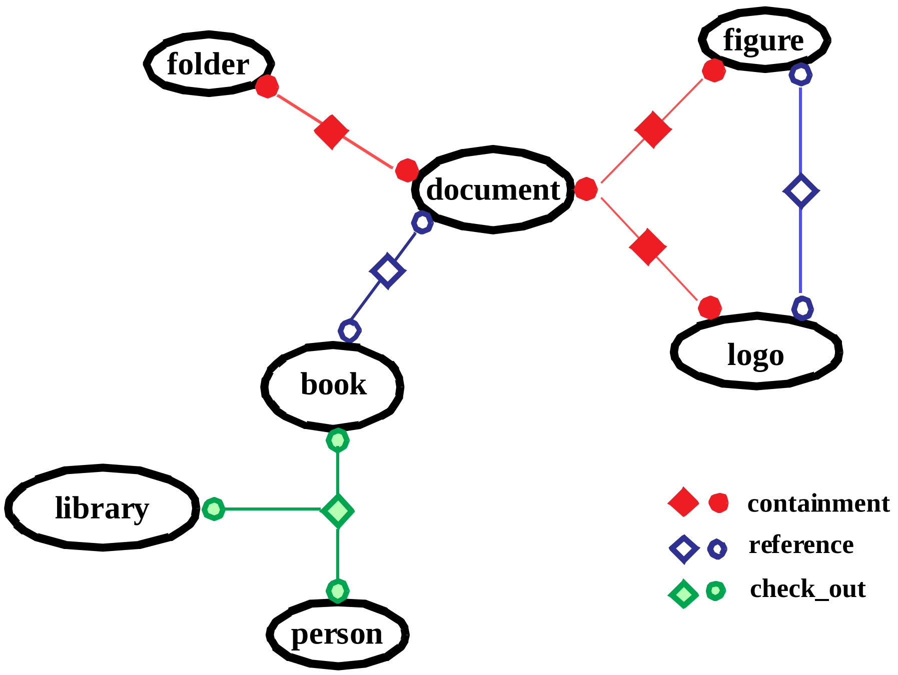

Figure 9-3 An example graph of related objects.

Table 9-3 lists the interfaces to support graphs of related objects. Section 9 specifies the interfaces in detail.

Level Three: Specific Relationships

Containment and reference are two important relationships. The Relationship Service defines these two binary relationships.

Table 9-4 and Table 9-5 list the interfaces defining specific relationships. Section 9 specifies the interfaces in detail.

9.2.2 Hierarchy of Relationship Interface

The relationship interfaces are arranged into the interface hierarchy illustrated in Figure 9-4.

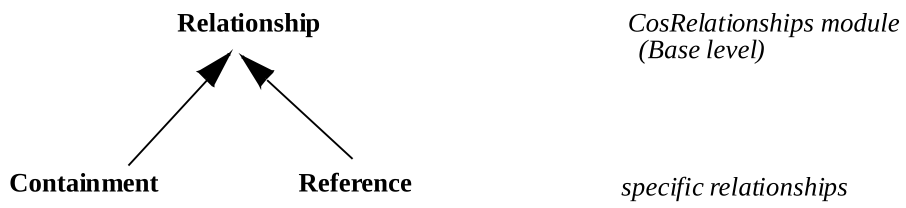

Figure 9-4 Relationship interface hierarchy

9.2.3 Hierarchy of Role Interface

The role interfaces are arranged into the interface hierarchy illustrated in Figure 9-5.

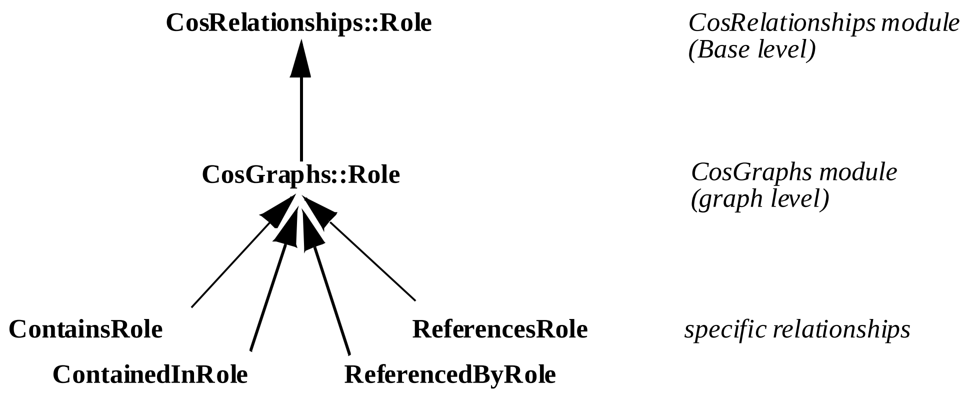

Figure 9-5 Role interface hierarchy

The *Role* interface defines operations to efficiently navigate relationships between related objects.

The *CosGraphs::Role* interface defines an operation to return the edges that involve the role. This is used by the traversal service defined at the graph level.

Finally, *ContainsRole*, *ContainedInRole*, *ReferencesRole* and *ReferencedByRole* are specific roles for two important relationships: containment and reference.

9.2.4 Interface Summary

The Relationship Service defines interfaces to support the functionality described in section 9.2.

Table 9-1 through Table 9-5 give high level descriptions of the Relationship Service interfaces. Sections 9 through 9 describe the interfaces in detail.

Table 9-1 Interfaces defined in the *CosObjectIdentity* module

Interface	**Purpose**	**IPrimary Clients**
CosObjectIdentity::		
IdentifiableObject	To determine if two objects are identical.	There are many clients. The graph level of the Relationship Service is one.

Table 9-2 Interfaces defined in the *CosRelationships* module

Interface	**Purpose**	**Primary Clients**
CosRelationships::		
Relationship	Represents an instance of a relationship type.	Clients that navigate between related objects.
RelationshipFactory	Supports the creation of relationships.	Clients establishing relationships.
Role	Defines navigation operations for relationships. Implements type and cardinality constraints.	Clients that navigate between related objects. Relationship factories.
RoleFactory	Supports the creation of roles.	Objects participating in relationships.
RelationshipIterator	Iterates the relationships in which a particular role object participates.	Clients that navigate relationships.

Table 9-3 Interfaces defined in the *CosGraphs* module

Interface	Purpose	Primary Client(s)
CosGraphs::		
Traversal	Defines an operation to traverse a graph, given a starting node and traversal criteria.	Clients that want a standard service to traverse graphs.
TraversalFactory	Supports the creation of a traversal object.	Clients that want a standard service to traverse graphs.
TraversalCriteria	Provides navigation behavior between nodes.	Traversal implementations.
Role	Extends the CosRelationships::Role interface to return edges	Clients that traverse graphs of related objects.
EdgeIterator	Returns additional edges from a role.	Clients that traverse graphs of related objects.
Node	Defines operations for a related object to reveal its roles.	Clients that traverse graphs of related objects.
NodeFactory	Supports the creation of nodes.	Clients that create nodes in graphs.

Table 9-4 Interfaces defined in the *CosContainment* module

Interface	Purpose	Primary Client(s)
CosContainment::		
Relationship	one-to-many relationship	Clients that depend on Containment relationship type.
ContainsRole	Represents an object that contains other objects.	Clients that navigate containment relationships between objects.
ContainedInRole	Represents an object that is contained in other objects.	Clients that navigate containment relationships between objects.

Table 9-5 Interfaces defined in the *CosReference* module

Interface	Purpose	Primary Clients
CosReference::		
Relationship	many-to-many relationship	Clients that depend on the reference relationship type.
ReferencesRole	Represents an object that references other objects.	Clients that navigate reference relationships between objects.
ReferencedByRole	Represents an object that is referenced by other objects.	Clients that navigate reference relationships between objects.

9.3 The Base Relationship Model

The base level of the Relationship Service defines interfaces that support relationships between two or more CORBA objects. Objects that participate in a relationship are called related objects. Relationships that share the same semantics form *relationship types*. A relationship is an instance of a relationship type and has an identity.

Each related object is connected with the relationship via a role. Roles are objects which characterize a related object‘s participation in a relationship type. Role types are used for expressing the role´s characteristics by an IDL interface. Cardinality represents the number of relationship instances connected to a role. Degree represents the number of roles in a relationship. All characteristics are expressed by corresponding IDL interfaces. Relationship and role types are built by subtyping the Relationship and Role interfaces.

Figure 9-6 gives a graphical representation of a simple relationship type. It illustrates that documents reference books. Documents are in the *ReferencesRole* and books are in the *ReferencedByRole*. Documents, reference, the roles and books are all types; there are interfaces (written in OMG IDL) for all five.

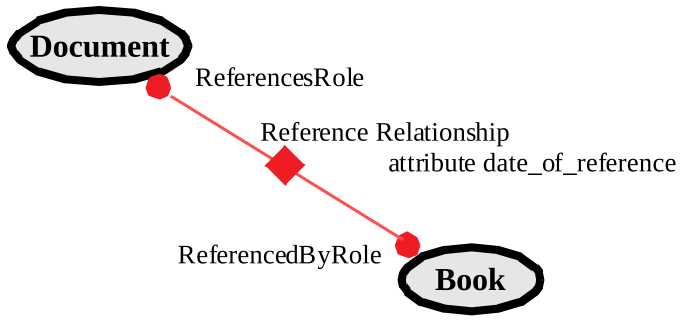

Figure 9-6 Simple relationship type: documents reference books

Figure 9-7, on the other hand, gives a graphical representation of an instance of a relationship type. It illustrates that “my document”, an instance of Document, references “War and Peace”, an instance of Book.[1]

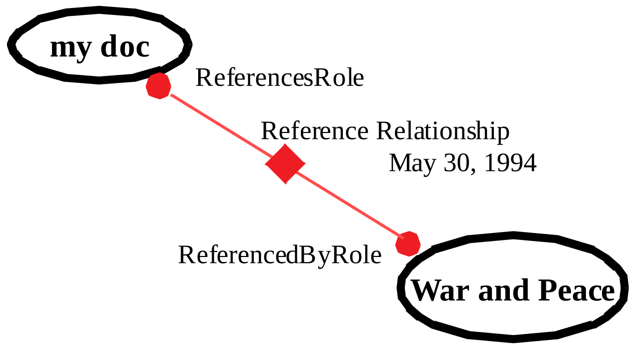

Figure 9-7 Simple relationship instance: my document references the book “War and Peace“

9.3.1 Relationship Attributes and Operations

Relationships may have attributes and operations. For example, the reference relationship of Figure 9-6 has an attribute indicating the date the reference from the document to the book was established.

1. Most of the figures in this specification represent instances of related objects, roles and relationships. Figures describing object and relationship type are clearly marked.

Rationale

If relationships are not allowed to define attributes and operations, they will have to be assigned to one of the related objects. This approach is prone to misunderstandings and inconsistencies. The approach to define an artificial related object, which then carries the attributes, is equally unsatisfactory.

The date attribute of the example of Figure 9-7 is clearly an attribute of the relationship, not one of related objects. It cannot be an attribute of “my document” since “my document” can reference many books on different dates. Similarly, it cannot be an attribute of “War and Peace” since “War and Peace” can be referenced by many books on different dates.

9.3.2 Higher Degree Relationships

The Reference relationship in Figure 9-6 is a *binary* relationship; that is, it is defined by two roles. The Relationship Service can also support relationships with more than two roles. The fact that three or more related objects may be part of a relationship can be expressed directly by means of the same concept as in the binary case. The *degree* represents the number of roles in a relationship. The Relationship Service supports higher degree relationships, that is relationships with degree greater than two.

Figure 9-8 shows a ternary “check out” relationship between books, libraries and persons. The semantics of this relationship is that a person borrows a book from a library. The relationship also defines an attribute that indicates the date when the book is due to be returned by the person to the library.

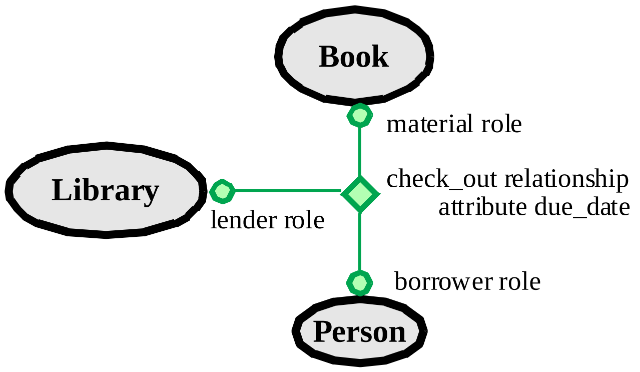

Figure 9-8 A ternary check-out relationship type between books, libraries and persons.

Rationale

The Relationship Service represents higher degree relationships directly. It clearly defines the number of expected related objects as well as other integrity constraints. It is more readable, more understandable and easier to enforce consistency constraints for related objects with a direct representation than with alternative representations that simulate higher degree relationships using a set of binary relationships. When simulating higher degree relationships, the relationship information is spread over multiple object and relationship type definitions, as are the corresponding integrity constraints.

Figure 9-9 shows an alternative representation of the ternary relationship from Figure 9-8 using binary relationships. Note that the first representation is not equivalent to that of Figure 9-8 since cardinalities and other integrity constraints cannot be expressed correctly in this alternative representation.

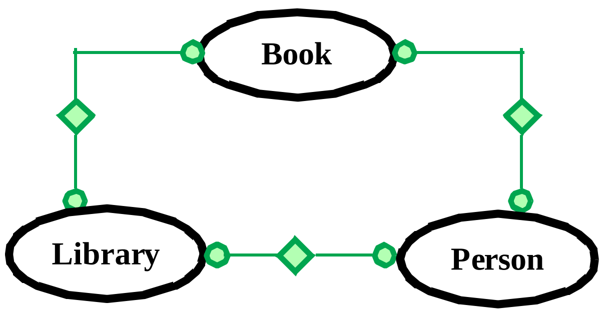

Figure 9-9 An unsatisfactory representation of the ternary check-out relationship using binary relationships.

Figure 9-10 illustrates a second alternative representation of the ternary relationship of Figure 9-8. It uses an additional (artificial) related object type. This representation is equivalent to Figure 9-8 if *Check-out* is constrained to participate in exactly one instance of each of the three binary relationship types. However, this alternative needs three relationship types and one additional related object type *(Check-out)* instead of only one relationship type, and therefore is much more complex and harder to capture when compared to the representation using one relationship type with degree 3.

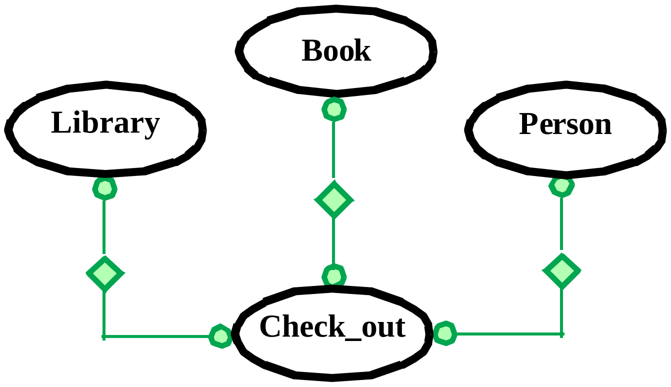

Figure 9-10 Another unsatisfactory representation

Since the Relationship Service supports higher order relationships directly, the user of the service need not resort to the unsatisfactory representations using binary relationships of Figure 9-9 and Figure 9-10.

9.3.3 Operations

The base level of the Relationship Service provides operations to:

- Create role and relationship objects
- Navigate relationships
- Destroy roles and relationships
- Iterate over the relationships in which a role participates

Creation

Roles are constructed independently using a role factory. Roles represent an existing related object that is passed as a parameter to the `RoleFactory::create` operation. When creating a new role object, the type of the related object can be checked by the factory. The minimum and maximum cardinality, e.g. the minimal and the maximal number of relationship instances to which the new role object may be connected, are indicated by attributes on the factory.

Figure 9-11 illustrates a newly created role.

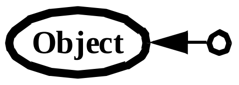

Figure 9-11 Creating a role for an object

A new relationship is created by passing a sequence of named roles to a factory for the relationship. The expected degree and role types for the new relationship are indicated by attributes on the factory. During the creation of the new relationship, the role types and the maximum cardinality can be checked. Duplicate role names are not allowed since the names are used to distinguish the roles in the scope of the relationship.

When creating a relationship, the factory creates “links” between the roles and the relationship using the `link` operation on the role.

Figure 9-12 illustrates a fully established binary relationship.[2]

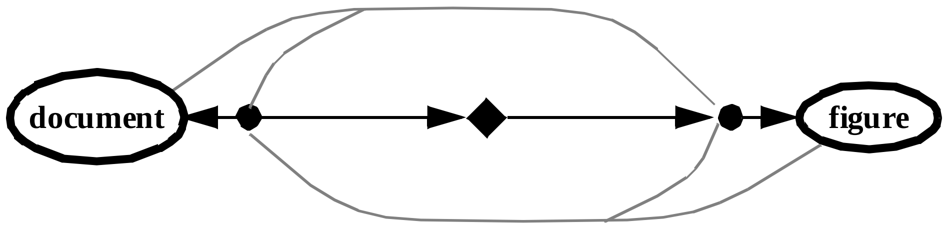

Figure 9-12 A fully established binary relationship

Navigation

Figure 9-12 illustrates the navigational functionality of a relationship. In particular,

- a relationship defines an attribute that indicates a read-only attribute that indicates the named roles of the relationship,
- a role defines a read-only attribute that indicates the related object that the role represents,
  - A role supports the `get_other_role` operation, that given a relationship object and a role name, returns the other role object,
  - A role supports the `get_other_related_object` operation, that given a relationship object and a role name, returns the related object that the named role represents in the relationship and
  - A role supports the `get_relationships` operation which returns the relationships in which the role participates.

Destruction

For both roles and relationship objects, the Relationship Services introduces a `destroy` operation. The destroy operation for relationship objects also destroys the links between the relationship and all of the role objects.

9.3.4 Consistency Constraints

For each role two cardinalities are defined: minimum and maximum.

- The minimum cardinality indicates the minimum number of relationship instances in which a role must participate.
- The maximum cardinality indicates the maximum number of relationship instances in which a role can participate.

Maximum cardinality constraint can be checked when relationships are created. Note that the relationship mechanism cannot, by itself, enforce the minimum cardinality constraint. However, a role can be asked explicitly if it meets its minimum cardinality constraint using the `check_minimum_cardinality` operation.

Type integrity is preserved by CORBA mechanisms because related objects, roles and relationships are instances of CORBA object types. Type constraints can be checked when roles and relationships are created.

2. Figure 9-12 represents *navigation functionality*; it does not necessarily represent stored object references. A variety of implementation strategies are described in section 9.3.5.

9.3.5 Implementation Strategies

9-12 illustrates the navigational functionality of a fully established binary relationship. There are a variety of implementation strategies possible. The `get_other_role` and the `get_other_related_object` operations can be:

- Implemented by caching object references to other roles and related objects, or
- Computed when needed using the relationship object.

The appropriate implementation strategy typically depends on distribution boundaries. If the roles and relationship objects are clustered, then only storing the values at the relationship object optimizes space. If, on the other hand, the roles and the related objects are clustered, caching object references to other roles and related objects at the roles allows the relationship to be efficiently navigated without involving a remote relationship object.

Role implementations that cache object references to other roles and related objects need not worry about updating the cache. Once the related objects and relationships are established, they cannot be changed.

9.3.6 The CosObjectIdentity Module

CORBA: Common Object Request Broker Architecture and Specification does not define a notion of object identity for objects. The Relationship Service requires object identity for the objects it defines. As such, the Relationship Service assumes the CosObjectIdentity module specified in Figure 9-13 . This is defined in a separate module; other Object Services may find this module to be generally useful.

```
module CosObjectIdentity {

    typedef unsigned long ObjectIdentifier;

    interface IdentifiableObject {
        readonly attribute ObjectIdentifier constant_random_id;
        boolean is_identical (
            in IdentifiableObject other_object);
    };

};
```

Figure 9-13 The CosObjectIdentity Module

The IdentifiableObject Interface

Objects that support the *IdentifiableObject* interface implement an attribute of type *ObjectIdentifier* and the `is_identical` operation. This mechanism provides an efficient and convenient method of supporting object identity in a heterogeneous CORBA-based environment.

constant_random_id

```
readonly attribute ObjectIdentifier constant_random_id;
```

Objects supporting the *IdentifiableObject* interface define an attribute of type *ObjectIdentifier*. The value of the attribute must not change during the lifetime of the object.

A typical client use of this attribute is as a key in a hash table. As such, the more randomly distributed the values are, the better.

The value of this attribute is not guaranteed to be unique; that is, another identifiable object can return the same value. However, if objects return different identifiers, clients can determine that two identifiable objects are ***not*** identical.

To determine if two identifiable objects ***are*** identical, the `is_identical` operation must be used.

is_identical

```
boolean is_identical (
    in IdentifiableObject other_object);
```

The `is_identical` operation returns *true* if the object and the `other_object` are identical. Otherwise, the operation returns *false*.

9.3.7 The CosRelationships Module

The CosRelationships module defines the interfaces of the base level Relationship Service. In particular, it defines

- *Relationship* and *Role* interfaces to represent relationships and roles,
- *RelationshipFactory* and *RoleFactory* interfaces to create relationships and roles
- *RelationshipIterator* interface to enumerate the relationships in which a role participates

The CosRelationships module is shown in Figure 9-14.

```
#include <ObjectIdentity.idl>

module CosRelationships {

    interface RoleFactory;
    interface RelationshipFactory;
    interface Relationship;
    interface Role;
    interface RelationshipIterator;

    typedef Object RelatedObject;
    typedef sequence<Role> Roles;
    typedef string RoleName;
    typedef sequence<RoleName> RoleNames;

    struct NamedRole {RoleName name; Role aRole;};
    typedef sequence<NamedRole> NamedRoles;

    struct RelationshipHandle {
        Relationship the_relationship;
        CosObjectIdentity::ObjectIdentifier constant_random_id;
    };
    typedef sequence<RelationshipHandle> RelationshipHandles;

    interface RelationshipFactory {
        struct NamedRoleType {
            RoleName name;
            ::CORBA::InterfaceDef named_role_type;
        };
        typedef sequence<NamedRoleType> NamedRoleTypes;
        readonly attribute ::CORBA::InterfaceDef relationship_type;
        readonly attribute unsigned short degree;
        readonly attribute NamedRoleTypes named_role_types;
        exception RoleTypeError {NamedRoles culprits;};
        exception MaxCardinalityExceeded {
            NamedRoles culprits;};
        exception DegreeError {unsigned short required_degree;};
        exception DuplicateRoleName {NamedRoles culprits;};
        exception UnknownRoleName {NamedRoles culprits;};

        Relationship create (in NamedRoles named_roles)
            raises (RoleTypeError,
                MaxCardinalityExceeded,
                DegreeError,
                DuplicateRoleName,
                UnknownRoleName);
    };
```

Figure 9-14 The CosRelationships Module

```
interface Relationship :
        CosObjectIdentity::IdentifiableObject {
    exception CannotUnlink {
        Roles offending_roles;
    };
    readonly attribute NamedRoles named_roles;
    void destroy () raises(CannotUnlink);
};

interface Role {
        exception UnknownRoleName {};
        exception UnknownRelationship {};
        exception RelationshipTypeError {};
        exception CannotDestroyRelationship {
            RelationshipHandles offenders;
        };
        exception ParticipatingInRelationship {
            RelationshipHandles the_relationships;
        };
        readonly attribute RelatedObject related_object;
    RelatedObject get_other_related_object (
            in RelationshipHandle rel,
            in RoleName target_name)
        raises (UnknownRoleName,
            UnknownRelationship);
    Role get_other_role (in RelationshipHandle rel,
            in RoleName target_name)
        raises (UnknownRoleName, UnknownRelationship);
    void get_relationships (
            in unsigned long how_many,
            out RelationshipHandles rels,
            out RelationshipIterator iterator);
    void destroy_relationships()
        raises(CannotDestroyRelationship);
    void destroy() raises(ParticipatingInRelationship);
    boolean check_minimum_cardinality ();
    void link (in RelationshipHandle rel,
            in NamedRoles named_roles)
        raises(RelationshipFactory::MaxCardinalityExceeded,
            RelationshipTypeError);
    void unlink (in RelationshipHandle rel)
        raises (UnknownRelationship);
};

interface RoleFactory {
    exception NilRelatedObject {};
    exception RelatedObjectTypeError {};
    readonly attribute ::CORBA::InterfaceDef role_type;
```

Figure 9-14 The CosRelationships Module *(Continued)*

```
        readonly attribute unsigned long max_cardinality;
        readonly attribute unsigned long min_cardinality;
        readonly attribute sequence
            <::CORBA::InterfaceDef> related_object_types;
        Role create_role (in RelatedObject related_object)
            raises (NilRelatedObject, RelatedObjectTypeError);
    };

    interface RelationshipIterator {
        boolean next_one (out RelationshipHandle rel);
        boolean next_n (in unsigned long how_many,
                out RelationshipHandles rels);
        void destroy ();
    };

};
```

Figure 9-14 The CosRelationships Module *(Continued)*

Example of Containment Relationships

The example of Figure 9-15 is referred to throughout the following sections to describe roles and relationships. The figure represents two binary, one-to-many containment relationships between a document and a figure and a logo.

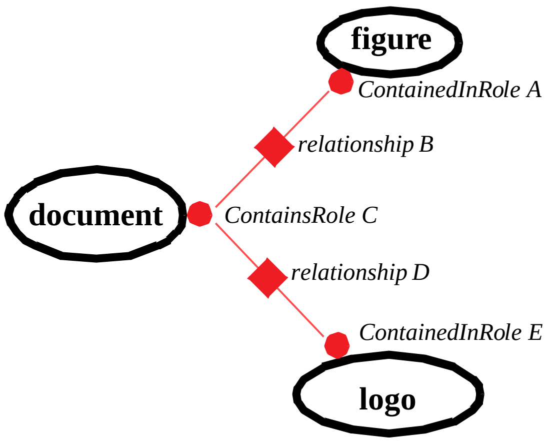

Figure 9-15 Two binary one-to-many containment relationships.

The RelationshipFactory Interface

The *RelationshipFactory* interface defines an operation for creating an instance of a relationship among a set of related objects. The factory also defines two attributes that specify the degree and role types of the relationships it creates.

Creating a Relationship

```
Relationship create (in NamedRoles named_roles)
    raises (RoleTypeError,
        MaxCardinalityExceeded,
        DegreeError,
        DuplicateRoleName,
        UnknownRoleName);
```

The `create` operation creates a new instance of a relationship. The factory is passed a sequence of named roles that represent the related objects in the newly created relationship. The factory, in turn, informs the roles about the new relationship using the `link` operation described in section .

Roles implement maximum cardinality constraints. A role may refuse to participate in a new relationship because it would violate a cardinality constraint. In such a case, the `MaxCardinalityExceeded` exception is raised and the offending roles are returned in the exception.

The number of roles passed to the `create` operation must be the same as the value of the `degree` attribute. If not, the `DegreeError` exception is raised.

Role names are used to associate each actual role object with one of the formal roles expected by the relationship to be created.

The set of role names passed to the `create` operation must be the same as the set of role names in the factory's `named_role_types` attribute. If not, the `UnknowRoleName` exception is raised, and the unrecognized names are returned in the exception. The sequence order of the `named_roles` parameter and the sequence order of the `named_role_types` need not correspond.

The type of each role passed to the `create` operation must be of the same type as the type indicated for the corresponding role name in the `named_role_types` attribute. If not, the `RoleTypeError` is raised and the offending roles are returned in the exception.

The names of the roles passed to the `create` operation must be unique within the scope of this relationship type. If not, the `DuplicateRoleName` exception is raised.

Example of Figure 9-15

The document and the figure were related, that is relationship B was created, by passing roles A and C to the `create` operation of the relationship factory. Similarly, the document and the logo were related by passing roles C and E to the relationship factory for relationship D.

Determining the Created Relationship's Type

```
readonly attribute ::CORBA::InterfaceDef relationship_type;
```

The relationship created by a factory may be a subtype of the *Relationship* interface. The r`relationship_type` attribute indicates the actual types of the relationships created by the factory.

Determining the Degree of a Relationship Type

```
readonly attribute unsigned short degree;
```

The `degree` attribute indicates the number of roles for the relationships created by the factory.

Example of Figure 9-15

The relationship factory for containment has a degree attribute whose value is 2 because containment is a binary relationship.

Determining Names and Types of the Roles of a Relationship Type

```
readonly attribute NamedRoleTypes named_role_types;
```

The `named_role_types` attribute indicates the required names and types of roles for the relationships created by the factory. NamedRoleTypes are defined as structures where the role type is given by the `CORBA::InterfaceDef` for the role objects.

Example of Figure 9-15

The relationship factory for containment has an attribute whose value is a sequence of two CORBA::InterfaceDefs: one for ContainsRole and one for ContainedInRole.

The Relationship Interface

The *Relationship* interface defines an attribute whose value is the named roles of the relationship and an operation to destroy the relationship.

Determining the Roles of a Relationship and Their Names

```
readonly attribute NamedRoles named_roles;
```

The `named_roles` attribute returns the roles of the relationship. The roles have the names that were indicated in the `create` operation defined by the *RelationshipFactory* interface.

Example of Figure 9-15

Relationship B has an attribute whose value is a sequence <“A”,InterfaceDef for ContainedInRole; “C”, InterfaceDef for ContainsRole>. Similarly, relationship D has an attribute whose value is a sequence <“E”, InterfaceDef for ContainedInRole; “C”, InterfaceDef for ContainsRole>.

Destroying a Relationship

```
void destroy () raises(CannotUnlink);
```

The `destroy` operation destroys the relationship between the objects. The roles are unlinked by the relationship implementation before it is destroyed. If roles cannot be unlinked, the `CannotUnlink` exception is raised and the roles that could not be unlinked are returned in the exception.

Example of Figure 9-15

If `destroy` is requested of relationship B, the `unlink` operation is requested of both roles A and C and the relationship B is destroyed.

The Role Interface

The *Role* interface defines operations to:

- navigate the relationship from one role to another,
- enumerate the relationships in which the role participates,
- destroy all relationships in which the role participates,
- link a role to a newly created relationship and
- unlink a role in the destruction process of a relationship and
- destroy the role itself,

Determining the Related Object That a Role Represents

```
readonly attribute RelatedObject related_object;
```

The `related_object` attribute indicates the related object that the role represents. The related object that the role represents is specified as a parameter to the `create` operation defined by the *RoleFactory* interface.

Getting Another Related Object

```
RelatedObject get_other_related_object (
        in RelationshipHandle rel,
        in RoleName target_name)
    raises (UnknownRoleName,
        UnknownRelationship);
```

The `get_other_related_object` operation navigates the relationship `rel` to the related object represented by the role named `target_name`.

If the role does not know about a role named `target_name`, the `UnknownRoleName` exception is raised. If the role does not know about the relationship rel, the `UnknownRelationship` exception is raised.

Example of Figure 9-15

Assuming role A is named “A”, requesting `get_other_related_object(B,”A”)` of role C returns the figure. On the other hand, requesting `get_other_related_object(D,”E”)` of role C returns the logo.

Getting Another Role

```
Role get_other_role (in RelationshipHandle rel,
        in RoleName target_name)
    raises (UnknownRoleName, UnknownRelationship);
```

The `get_other_role` operation navigates the relationship `rel` to the role named `target_name`. The role is returned.

If the role does not know about a role named `target_name` for the relationship `rel`, the `UnknownRoleName` exception is raised. If the role does not know about the relationship rel, the `UnknownRelationship` exception is raised.

Example of Figure 9-15

Assuming role A is named “A”, requesting `get_other_role(B,”A”)` of role C returns role A. On the other hand, requesting `get_other_role(D,”E”)` of role C returns role E.

Getting All Relationships in Which a Role Participates

```
void get_relationships (
       in unsigned long how_many,
       out RelationshipHandles rels,
       out RelationshipIterator iterator);
```

The `get_relationships` operation returns the relationships in which the role participates.

The size of the list is determined by the `how_many` argument. If there are more relationships than specified by the `how_many` argument, an iterator is created and returned with the additional relationships. If there are no more relationships, a nil object reference is returned for the iterator. (The *RelationshipIterator* interface is a standard iterator described in the next section.)

Example of Figure 9-15

Requesting `get_relationships` on role C would return the relationships B and D.

Destroying All Relationships in Which a Role Participates

```
void destroy_relationships()
    raises(CannotDestroyRelationship);
```

The `destroy_relationships` operation destroys all relationships in which the role participates.

The `destroy_relationships` operation is semantically equivalent to requesting destroy of each relationship in which the role participates. The operation is not required to be implemented in that fashion.

If the `destroy_relationships` operation cannot destroy one of the relationships, then the `CannotDestroyRelationship` exception is raised and the relationships that could not be destroyed are returned in the exception.

Example of Figure 9-15

Requesting `destroy_relationships` of role A causes relationship B to be destroyed. On the other hand, requesting `destroy_relationships` of role C causes relationships B and D to be destroyed.

Destroying a Role

```
void destroy() raises(ParticipatingInRelationship);
```

The `destroy` operation destroys the role. The role must not be participating in any relationships. If it is, the ParticipatingInRelationship exception is raised and the relationships in which the role participates are returned in the exception.

Example of Figure 9-15

Requesting `destroy_role` of role A destroys relationship B and role A.

Checking Minimum Cardinality of a Role

```
boolean check_minimum_cardinality ();
```

The `check_minimum_cardinality` operation returns *true* if a role satisfies its minimum cardinality constraints. Otherwise, the operation returns *false*.

Example of Figure 9-15

Requesting `check_minimum_cardinality` of role A would return true since it is participating in relationship B.

Linking a Role in a Newly Created Relationship

```
void link (in RelationshipHandle rel,
        in NamedRoles named_roles)
    raises(RelationshipFactory::MaxCardinalityExceeded,
        RelationshipTypeError);
```

Note – The `link` operation is not intended for general purpose clients that create, navigate and destroy relationships. Instead, it is an operation intended for implementations of the relationship factory `create` operation.

The `link` operation informs the role that a new relationship is being created. The role is passed a relationship and a set of named roles that represent related objects in the relationship.

A role can have a maximum cardinality, that is it may limit the number of relationships in which it participates. If the `link` request would cause the maximum to be exceeded, the `MaxCardinalityExceeded` exception is raised. If the type of the relationship does not agree with the relationship type that the role expects, the `RelationshipTypeError` exception is raised.

Example of Figure 9-15

When creating relationship B, the factory for B requested the link (B, A,C) operation on roles A and C. This allows roles A and C to support the navigation and administration operations for relationship B.

Removing a Role from a Relationship

```
void unlink (in RelationshipHandle rel)
   raises (UnknownRelationship);
```

Note – The `unlink` operation is not intended for general purpose clients that create, navigate and destroy relationships. Instead, it is an operation intended for implementations of the relationship `destroy` operation.

The `unlink` operation causes the role to delete its record of the relationship.

If the relationship passed as an argument is unknown to the role, the `UnknownRelationship` exception is raised.

Example of Figure 9-15

The implementation of the `destroy` operation on relationship B requests `unlink(B)` of roles A and C. This causes roles A and C to forget their participation in relationship B.

The RoleFactory Interface

The *RoleFactory* interface defines attributes describing the roles that it creates and a single operation to create a role.

Creating a Role

```
Role create_role (in RelatedObject related_object)
      raises (NilRelatedObject, RelatedObjectTypeError);
```

The `create_role` operation creates a role for the related object passed as a parameter.

A role must represent a related object. If a nil object reference is passed to the factory for the related object, the `NilRelatedObject` exception is raised.

Role factories can restrict the type of objects the roles they create will represent. If the interface of the related object does not conform, the `RelatedObjectTypeError` exception is raised.

Example of Figure 9-15

Clients that created roles A, C and E used the `create` operation of factories that support the *RoleFactory* interface.

Determining the Created Role's Type

```
readonly attribute ::CORBA::InterfaceDef role_type;
```

The role created by a factory may be a subtype of the *Role* interface. The `role_type` attribute indicates the actual types of the roles created by the factory.

Determining the Maximum Cardinality of a Role

```
readonly attribute unsigned long max_cardinality;
```

The `max_cardinality` attribute indicates the maximum number of relationships in which a role (created by the factory) participates.

Example of Figure 9-15

The factory for role A returns 1, since a *ContainedIn* role can be in no more than one relationship. Attempts to add role A to more than one relationship result in `MaxCardinalityExceeded` exceptions. (See the `create` operation of the *RelationshipFactory* interface and the `link` operation of the *Role* interface.)

Determining the Minimum Cardinality of a Role

```
readonly attribute unsigned long min_cardinality;
```

The `min_cardinality` attribute indicates the minimum number of relationships in which a role (created by the factory) participates.

Note, that unlike maximum cardinality, minimum cardinality cannot be enforced since roles will be below their minimum during relationship construction. Roles do support the `check_minimum_cardinality` operation to report if they are below their minimum.

Example of Figure 9-15

The factory for role A returns 1, since a *ContainedIn* role should be in one relationship.

Determining the Related Object Types for a Role

```
readonly attribute sequence
   <::CORBA::InterfaceDef> related_object_types;
```

The factory creates roles that represent related objects in relationships. The related objects must support at least one of the interfaces indicated by the `related_object_type` attribute.

Example of Figure 9-15

The factory for role C returns the CORBA::InterfaceDef for a document.

The RelationshipIterator Interface

The *RelationshipIterator* interface is returned by the `get_relationships` operation defined by the *Role* interface. It allows clients to iterate through any additional relationships in which the role participates.

next_one

```
boolean next_one (out RelationshipHandle rel);
```

The `next_one` operation returns the next relationship; if no more relationships exist, it returns *false*.

next_n

```
boolean next_n (in unsigned long how_many,
       out RelationshipHandles rels);
```

The `next_n` operation returns at most the requested number of relationships; if no more relationships exist, it returns *false*.

destroy

```
void destroy ();
```

The `destroy` operation destroys the iterator.

9.4 Graphs of Related Objects

When objects are related using the Relationship Service, *graphs of related objects* are formed. This section focuses on how the Relationship Service supports graphs of related objects. We first describe the graph architecture supported by the service, describe support for traversing the graph and implementing compound operations and then specify the *CosGraphs* module in detail.

Graphs are important for distributed, object-oriented applications. A few examples of graphs are:

Distributed Desktops

Folders and objects are connected together. Folders contain some objects and reference others. Folders may contain or reference other folders. The objects are distributed; they span multiple machines. The distributed desktop is a distributed graph.

Composed Applications

Applications are built out of existing objects that are connected together. An example of such a composed application is a shared white board. The composed application is a graph.

User Interface Hierarchies

Presentation objects visualize semantic objects for users. Presentations contain other presentation objects. For example, a window might contain a button. The user interface hierarchy is a graph.

Compound Documents

A compound document architecture allows graphics, animation, sound, video, etc. to be connected together to give the user the impression of a single document. The compound document is a graph.

9.4.1 Graph Architecture

A graph is a set of nodes and a set of edges, involving those nodes. Nodes are related objects that support the *Node* interface and edges are represented by the relationships that relate nodes.

Figure 9-3 on page 9-9 illustrates an example of a graph.

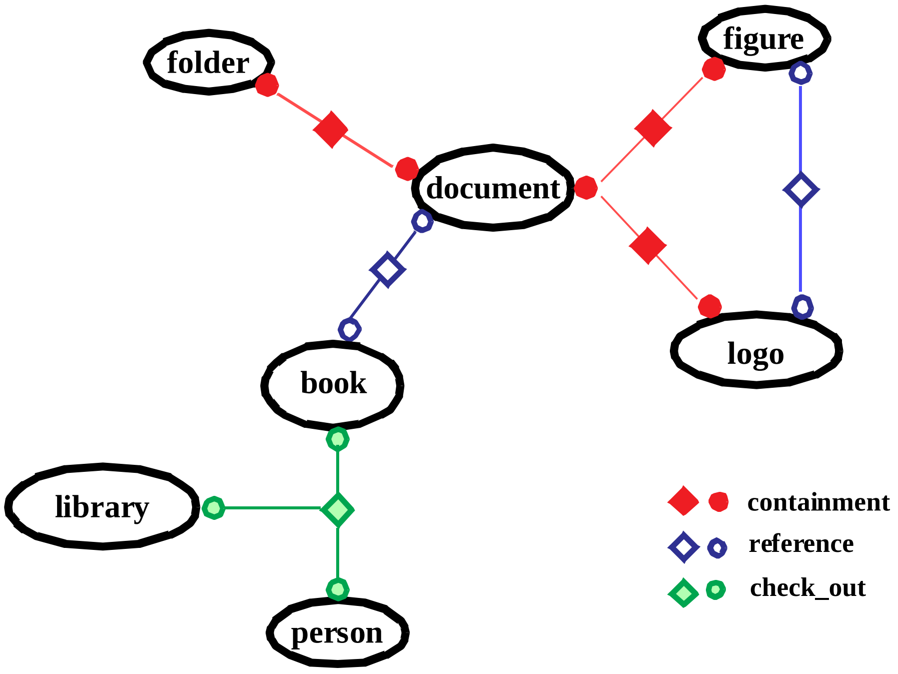

Figure 9-16 An example graph of related objects.

The folder, book, document, figure, library, person and logo are nodes in the graph. The edges of the graph are represented by the relationships:

- containment: the folder and document,
- containment: the document and the figure
- containment: the document and the logo
- reference: the figure and the logo
- reference: the document and the book,
- check_out: the book, the library and the person

The graph architecture supports multiple kinds of relationships. For example, in Figure 9-3, there are containment, reference and check_out relationships. The small circles depict roles for a reference relationship, the solid circles depict roles for a containment relationship and the shaded circles represent the roles of the check_out relationship.

A node can participate in more than one kind of relationship and thus have more than one role. In the example the document has three kinds of roles:

- The *ContainsRole*
- The *ContainedInRole*
- The *ReferencesRole*

Nodes

Nodes are identifiable objects that support the *Node* interface. Nodes collect roles of a related object and the related object itself. A node enables standard traversals of graphs of related objects because it supports the following:

- A readonly attribute defining all of its roles
- An operation allowing roles of a particular type to be returned
- Operations to add and remove roles

The *Node* interface can be inherited by related objects or an object implementing the *Node* interface can be instantiated and interposed in front of related objects. Interposition is particularly useful in these cases:

- When connecting immutable objects, which are objects that are not aware of the Relationship Service
- In order to traverse graphs of related objects without activating the related objects

As such, the *Node* interface defines an attribute whose value is the related object it represents.

9.4.2 Traversing Graphs of Related Objects

The Relationship Service defines a traversal object that, given a starting node, produces a sequence of directed edges of the graph. A directed edge corresponds to a relationship. In particular, it consists of:

- An instance of a relationship,
- A starting node and a starting named role of the edge to indicate direction and
- A sequence containing the remaining nodes and named roles. For binary relationships, there is a single remaining node and role. For n-ary relationships, there are n-1 remaining nodes and roles.

The traversal object works like an iterator, where directed edges are the items being returned.

The traversal object, the nodes and the roles cooperate in traversing the graph. Through the operations of the *Node* interface, the node reveals its roles to the traversal object. Through the operations of the *CosGraphs::Role* interface, a role reveals its directed edges to other nodes. (The *CosGraphs::Role* interface defines an operation allowing a role to reveal directed edges.)

In traversing a graph, the traversal object must detect and represent cycles, and determine the relevant nodes and edges.

Detecting and Representing Cycles

In order to terminate, a traversal must be able to detect a cycle in the graph. In the example of 9-3, the document, the figure, and the logo form a cycle.

To detect cycles in the graph, the traversal object depends on the fact that nodes are identifiable objects, that is they support the *IdentifiableObject* interface defined in section 9.3.6.

To represent cycles in the graph, the traversal object defines a scope of identifiers for the nodes and relationships in the graph. That is, a given traversal *assigns* identifiers to the nodes and relationships that are guaranteed to be unique within the scope of the traversal.

Determining the Relevant Nodes and Edges

A traversal begins at the starting node, emits directed edges and *may* continue to other related nodes. The traversal object is programmable in the criteria it uses for determining the edges to emit and the nodes to visit. The traversal object depends on a "call-back" object supporting the *TraversalCriteria* interface.

Given a node, the traversal criteria computes a sequence of directed edges to include in the traversal. For each edge, the traversal criteria can indicate whether the traversal should continue to an adjacent node. Based on the results of the traversal criteria, the traversal object emits edges and visits other nodes. The process continues until there are no more edges to emit and no more nodes to visit.

Three standard traversal modes are defined to allow clients flexibility in controlling the search order: *depth first*, *breadth first*, and *best first*. In order to understand the differences between the modes, consider that the traversal maintains an ordered list of the edges which have been produced by visiting nodes. This list initially contains the edges which result from visiting the root node. In each iteration the first edge is removed from the list to be returned and its destination nodes are visited. Depending upon the traversal mode, these edges are: inserted in the beginning of the list (depth first), appended to the end of the list (breadth first), or inserted into the list which is sorted by the edge's weight (best first).

9.4.3 Compound Operations

Traversal objects are especially important in implementing compound operations on graphs of related objects. By compound operations, we mean operations that apply to some subset of the nodes and edges in the graph. Examples of compound operations include operations, such as copy, move, remove, externalize, print, and so forth.

Note – The Relationship Service defines a framework for compound operations but does not define specific compound operations. The Life Cycle and the Externalization Service specifications define compound operations that depend on the Relationship Service.

A compound operation may be implemented either in one or two passes. A compound operation implemented in one pass traverses the graph itself and applies the operation as it proceeds.

A compound operation implemented in two passes uses the traversal object defined by the Relationship Service to determine the relevant nodes and detect and represent cycles. The second pass simply applies the operation to the results of the first pass.

A compound operation implemented in two passes provides a *TraversalCriteria* object for the traversal service.

9.4.4 An Example Traversal Criteria

Consider a traversal of a graph with a traversal criteria object that uses propagation values defined by the relationships to determine whether to emit an edge and whether to proceed to another node. The traversal criteria is given a node by the traversal. The traversal criteria then requests propagation values from each of the node's roles.

Figure 9-17 illustrates a traversal of a graph using a traversal criteria for a compound `copy` operation. Using the `propagation_for` operation defined by *CompoundLifeCycle::Role* interface, the traversal criteria obtains the propagation value for the copy operation from each of the node's roles.

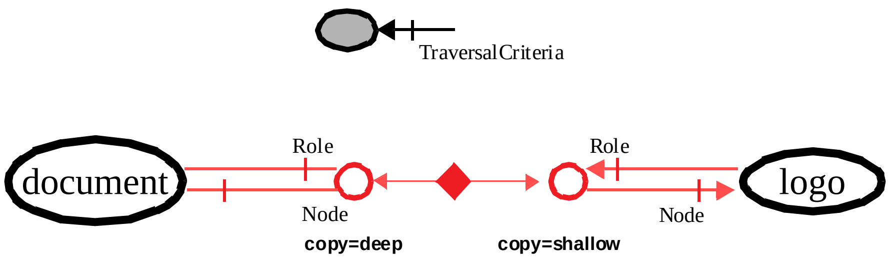

Figure 9-17 A traversal of a graph for compound copy operation.

Propagation

Compound operations may propagate from one node to another depending on the semantics of the relationship between the nodes. The propagation semantics of a relationship depend on the direction the relationship is being traversed. A propagation value is either *deep*, *shallow*, *inhibit* or *none*.

Deep means that the operation is applied to the node, to the relationship and to the related objects. In the example of Figure 9-17, the propagation value for the copy operation is deep from the document to the logo; the copy propagates from the document to the logo across the containment relationship. The traversal criteria for copy that encounters a deep propagation value would instruct the traversal object to emit the edge and visit the logo.

Shallow means that the operation is applied to the relationship but not to the related objects. In the example of Figure 9-17, the propagation value for the copy operation from the logo to the document is shallow. The traversal criteria for copy that encounters a shallow propagation value would instruct the traversal object to emit the edge but the document is not visited.

None means that the operation has no effect on the relationship and no effect on the related objects. A traversal criteria that encounters a none propagation value would not return any edges and related nodes are not visited.

Figure 9-18 summarizes how deep, shallow and node propagation values affect nodes, roles and relationships.

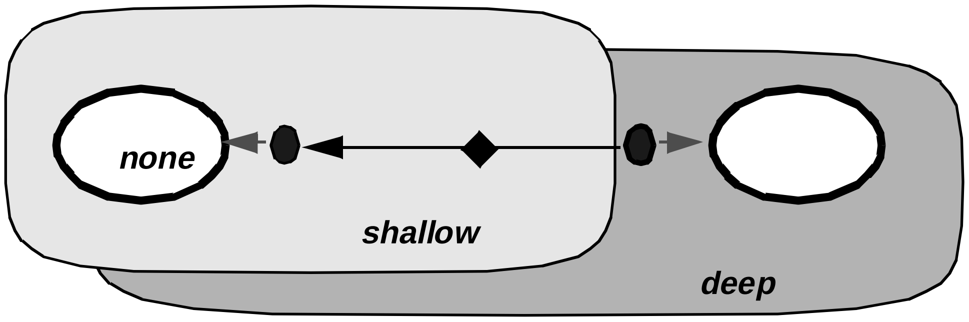

Figure 9-18 How deep, shallow and none propagation values affect nodes, roles and relationships.

Inhibit means that the operation should not propagate to the node via any of the node's roles. Inhibit is particularly meaningful for the remove operation to provide so-called "existence-ensuring relationships".

For more discussion of propagation values, see [1].

9.4.5 The CosGraphs Module

The CosGraphs module defines the support for graphs of related objects. It defines the following interfaces:

- *TraversalFactory* interface for creating traversal objects
- *Traversal* interface for enumerating directed edges of a graph,
- *TraversalCriteria* "call-back" interface to allow programmability of the traversal object
- *Node* interface for collecting the roles of a related object
- *NodeFactory* interface for creating nodes
- *Role* interface to support traversals

The CosGraphs module is shown in Figure 9-14.

```
#include <Relationships.idl>
#include <ObjectIdentity.idl>

module CosGraphs {

        interface TraversalFactory;
        interface Traversal;
        interface TraversalCriteria;
        interface Node;
        interface NodeFactory;
        interface Role;
        interface EdgeIterator;

        struct NodeHandle {
            Node the_node;
            ::CosObjectIdentity::ObjectIdentifier constant_random_id;
        };
        typedef sequence<NodeHandle> NodeHandles;

        struct NamedRole {
            Role the_role;
            ::CosRelationships::RoleName the_name;
        };
        typedef sequence<NamedRole> NamedRoles;

        struct EndPoint {
            NodeHandle the_node;
            NamedRole the_role;
        };
        typedef sequence<EndPoint> EndPoints;

        struct Edge {
            EndPoint from;
            ::CosRelationships::RelationshipHandle the_relationship;
            EndPoints relatives;
        };
        typedef sequence<Edge> Edges;

        enum PropagationValue {deep, shallow, none, inhibit};
        enum Mode {depthFirst, breadthFirst, bestFirst};

        interface TraversalFactory {
            Traversal create_traversal_on (
                    in NodeHandle root_node,
                    in TraversalCriteria the_criteria,
                    in Mode how);
        };
```

Figure 9-19 The CosGraphs Module

```
interface Traversal {
    typedef unsigned long TraversalScopedId;
    struct ScopedEndPoint {
        EndPoint point;
        TraversalScopedId id;
    };
    typedef sequence<ScopedEndPoint> ScopedEndPoints;
    struct ScopedRelationship {
        ::CosRelationships::RelationshipHandle
            scoped_relationship;
        TraversalScopedId id;
    };
    struct ScopedEdge {
        ScopedEndPoint from;
        ScopedRelationship the_relationship;
        ScopedEndPoints relatives;
    };
    typedef sequence<ScopedEdge> ScopedEdges;
    boolean next_one (out ScopedEdge the_edge);
    boolean next_n (in short how_many,
            out ScopedEdges the_edges);
    void destroy ();
};

interface TraversalCriteria {
    struct WeightedEdge {
        Edge the_edge;
        unsigned long weight;
        sequence<NodeHandle> next_nodes;
    };
    typedef sequence<WeightedEdge> WeightedEdges;
    void visit_node(in NodeHandle a_node,
            in Mode search_mode);
    boolean next_one (out WeightedEdge the_edge);
    boolean next_n (in short how_many,
            out WeightedEdges the_edges);
    void destroy();
};
```

Figure 9-19 The CosGraphs Module *(Continued)*

```
        interface Node: ::CosObjectIdentity::IdentifiableObject {
            typedef sequence<Role> Roles;
            exception NoSuchRole {};
            exception DuplicateRoleType {};

            readonly attribute ::CosRelationships::RelatedObject
                 related_object;
            readonly attribute Roles roles_of_node;
            Roles roles_of_type (
                    in ::CORBA::InterfaceDef role_type);
            void add_role (in Role a_role)
                    raises (DuplicateRoleType);
            void remove_role (in ::CORBA::InterfaceDef of_type)
                raises (NoSuchRole);
        };

        interface NodeFactory {
            Node create_node (in Object related_object);
        };

        interface Role : ::CosRelationships::Role {
            void get_edges ( in long how_many,
                    out Edges the_edges,
                    out EdgeIterator the_rest);
        };

        interface EdgeIterator {
            boolean next_one (out Edge the_edge);
            boolean next_n ( in unsigned long how_many,
                    out Edges the_edges);
            void destroy ();
        };

};
```

Figure 9-19 The CosGraphs Module *(Continued)*

The TraversalFactory Interface

The *TraversalFactory* interface creates traversal objects. The *Traversal* interface is used by clients that want to traverse graphs of related objects according to some traversal criteria.

create_traversal_on

```
Traversal create_traversal_on (
        in NodeHandle root_node,
        in TraversalCriteria the_criteria,
        in Mode how);
```

The `create_traversal_on` operation creates a traversal object starting at the `root_node`. The created traversal object uses the *TraversalCriteria* object to determine which directed edges to emit and which nodes to visit. The `mode` parameter indicates whether the traversal will proceed in a depth first, breadth first or best first fashion.

The Traversal Interface

Traversal objects iterate through `ScopedEdges` of the graph according to the traversal criteria and the mode established when the traversal was created. The traversal also defines a scope for the nodes and edges it returns; that is, it assigns identifiers to the nodes and edges it returns. The identifiers are unique within the scope of a given traversal. `ScopedEdges` are given by the following structure:

```
struct ScopedEdge {
    ScopedEndPoint from;
    ScopedRelationship the_relationship;
    ScopedEndPoints relatives;
};
typedef sequence<ScopedEdge> ScopedEdges;
```

A `ScopedEdge` consists of a distinguished scoped end point, a scoped relationship and a sequence of scoped end points. The distinguished scoped end point indicates the direction of the edge. The scoped end point consists of a node, a role, and an identifier for the node that is unique within the scope of the traversal.

next_one

```
boolean next_one (out ScopedEdge the_edge);
```

The `next_one` operation returns the next scoped edge; if no more scoped edges exist, it returns *false*.

next_n

```
boolean next_n (in short how_many,
       out ScopedEdges the_edges);
```

The `next_n` operation returns at most the requested number of scoped edges.

destroy

```
void destroy ();
```

The `destroy` operation destroys the traversal.

The TraversalCriteria Interface

The *TraversalCriteria* interface is used by the traversal object to determine which edges to emit and which nodes to visit from a given node. The traversal criteria behaves like an iterator of weighted edges. Weighted edges are given by the following structure:

```
struct WeightedEdge {
    Edge the_edge;
    unsigned long weight;
    sequence<NodeHandle> next_nodes;
};
typedef sequence<WeightedEdge> WeightedEdges;
```

A `WeightedEdge` consists of an edge, a weight and a sequence of nodes indicating if the traversal should continue to the nodes. The weight is only meaningful for the best first traversal.

next_one

```
boolean next_one (out WeightedEdge the_edge);
```

The `next_one` operation returns the next weighted edge; if no more weighted edges exist, it returns *false*.

next_n

```
boolean next_n (in short how_many,
        out WeightedEdges the_edges);
```

The `next_n` operation returns at most the requested number of weighted directed edges.

destroy

```
void destroy();
```

The `destroy` operation destroys the traversal criteria.

visit_node

```
void visit_node(in NodeHandle a_node,
        in Mode search_mode);
```

The `visit_node` operation establishes the node for which the traversal criteria will iterate and indicates the current search mode. As the traversal object traverses the graph, it visits nodes by requesting the `visit_node` operation of the traversal criteria, followed by `next_one/next_n` requests to obtain the outgoing edges from the node.

For depthFirst and breadthFirst modes, the weight field in the weighted edges is ignored. In the bestFirst mode, the weight value is utilized to order the traversal's edges list which is sorted by this value in ascending order.

If weighted edges from a previous node remain when `visit_node` is requested, the traversal criteria discards the previous edges.

The Node Interface

The *Node* interface defines operations that are useful in navigating graphs of related objects. In particular, it defines:

- Areadonly attribute giving all of the node's roles
- An operation allowing roles conforming to a particular type to be returned
- Operations to add and remove roles

Roles are distinguished in nodes in the OMG IDL of their interfaces.

A node cannot posses two roles where one role is a subtype of the other. This is precluded by the `add_role` operation.

A node can posses two or more roles that have a common supertype. The set of roles can be obtained by passing the common supertype to the `roles_of_type` operation.

related_object

```
readonly attribute ::CosRelationships::RelatedObject
     related_object;
```

The `related_object` attribute gives the related object that the node represents. This is useful when relating immutable objects.

roles_of_node

```
readonly attribute Roles roles_of_node;
```

The `roles_of_node` attribute gives all of the node's roles.

roles_of_type

```
Roles roles_of_type (
       in ::CORBA::InterfaceDef role_type);
```

The `roles_of_type` operation returns the node's roles that conform to the `role_type` parameter. A role conforms to `role_type` if it's interface is the same or is a subtype of `role_type`.

add_role

```
void add_role (in Role a_role)
    raises (DuplicateRoleType);
```

The `add_role` operation adds a role to the node. If the node posses a role of the same type, a supertype or a subtype of `a_role`, the `DuplicateRoleType` exception is raised.

remove_role

```
void remove_role (in ::CORBA::InterfaceDef of_type)
    raises (NoSuchRole);
```

The `remove_role` operation removes all the roles that conform to the `of_type` parameter. If no roles conform to the of_type parameter, the `NoSuchRole` exception is raised.

The NodeFactory Interface

The *NodeFactory* interface defines a single operation for creating nodes.

create_node

```
Node create_node (in Object related_object);
```

The `create_node` operation creates a node whose `related_object` attribute is initialized to the `related_object` parameter.

The Role Interface

The *CosGraphs::Role* interface extends the *CosRelationships::Role* interface with a single operation to return a role's view of it's relationships. The role's view of a relationship is given by the following `Edge` structure:

```
struct Edge {
    EndPoint from;
    ::CosRelationships::RelationshipHandle the_relationship;
    EndPoints relatives;
};
typedef sequence<Edge> Edges;
```

The edge structure is defined by an end point, a relationship and the other end points. The from end point is the role and its related object.

get_edges

```
void get_edges ( in long how_many,
       out Edges the_edges,
       out EdgeIterator the_rest);
```

The `get_edges` operation returns the edges in which the role participates.

The size of the list is determined by the `how_many` argument. If there are more edges than specified by the `how_many` argument, an iterator is created and returned. If there are no more edges, a nil object reference is returned for the iterator.

The EdgeIterator Interface

The *EdgeIterator* interface is returned by the `get_edges` operation defined by the *CosGraphs::Role* interface. It allows clients to iterate through any additional relationships in which the role participates.

next_one

```
boolean next_one (out Edge the_edge);
```

The `next_one` operation returns the next edge; if no more edges exist, it returns *false*.

next_n

```
boolean next_n ( in unsigned long how_many,
       out Edges the_edges);
```

The `next_n` operation returns at most the requested number of edges.

destroy

```
void destroy ();
```

The `destroy` operation destroys the iterator.

9.5 Specific Relationships

The Relationship Service defines two important relationships, *containment* and *reference* as part of its specification. The example used throughout this specification has been in terms of these two relationships.

9.5.1 Containment and Reference

Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment and reference are examples of relationships. However, since containment and reference are very common relationships, the Relationship Service defines them as standard.

Containment is defined by interfaces for a relationship and two roles: the *CosContainment::Relationship* interface, the *CosContainment::ContainsRole* interface, and the *CosContainment::ContainedInRole* interface. *Relationship* is a subtype of *CosRelationships::Relationship* and *ContainedInRole* and *ContainsRole* are subtypes of *CosGraphs::Role*.

Similarly, reference is defined by interfaces for a relationship and two roles: the *CosReference::Relationship* interface, the *CosReference::ReferencesRole* interface, and the *CosReference::ReferencedByRole* interface. *Relationship* is a subtype of *CosRelationships::Relationship* and *ReferencesRole* and *ReferencedByRole* are subtypes of *CosGraphs::Role*.

9.5.2 The CosContainment Module

The *CosContainment* module is shown in Figure 9-14.

```
#include <Graphs.idl>

module CosContainment {

    interface Relationship :
        ::CosRelationships::Relationship {};

    interface ContainsRole : ::CosGraphs::Role {};

    interface ContainedInRole : ::CosGraphs::Role {};

};
```

Figure 9-20 The CosContainment Module

The *CosContainment* module does not define new operations. It introduces new IDL types to represent containment. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
`relationship_type`	`CosContainment::Relationship`
`degree`	`2`
`named_role_types`	`“ContainsRole”,CosContainment::ContainsRole;` `“ContainedInRole”,CosContainment::ContainedInRole`

The CosRelationships::RelationshipFactory::`create` operation will raise `DegreeError` if the number of roles passed as arguments is not 2. It will raise `RoleTypeError` if the roles are not *CosContainment*::*ContainsRole* and *CosContainment*::*ContainedInRole*. It will raise `MaxCardinalityExceeded` if the *CosContainment*::*ContainedInRole* is already participating in a relationship.

RoleFactory attribute for ContainsRole	**value**
`role_type`	`CosContainment::ContainsRole`
`maximum_cardinality`	`unbounded`
`minimum_cardinality`	`0`
`related_object_types`	`CosGraphs::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosContainment::Relationship* interface.

RoleFactory attribute for ContainedInRole	**value**
`role_type`	`CosContainment::ContainedInRole`
`maximum_cardinality`	`1`
`minimum_cardinality`	`1`
`related_object_types`	`CosGraphs::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosContainment::Relationship* interface. The

CosRelationships::RoleFactory::`link` operation will raise `MaxCardinalityExceeded` if it is already participating in a containment relationship.

9.5.3 *The CosReference Module*

The *CosReference* module is given in Figure 9-21.

```
#include <Graphs.idl>

module CosReference {

    interface Relationship :
        ::CosRelationships::Relationship {};

    interface ReferencesRole : CosGraphs::Role {};

    interface ReferencedByRole : ::CosGraphs::Role {};

};
```

Figure 9-21 The CosReference Module

The *CosReference* module does not define new operations. It introduces new IDL types to represent reference. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
`relationship_type`	`CosReference::Relationship`
`degree`	`2`
`named_role_types`	`“ReferencesRole”,CosReference::ReferencesRole;` `“ReferencedByRole”,CoReference::ReferencedByRole`

The CosRelationships::RelationshipFactory::`create` operation will raise `DegreeError` if the number of roles passed as arguments is not 2. It will raise `RoleTypeError` if the roles are not *CosReference::ReferencesRole* and *CosReference*::*ReferencedByRole*.

RoleFactory attribute for ReferencesRole	**value**
`role_type`	`CosReference::ReferencesRole`
`maximum_cardinality`	`unbounded`
`minimum_cardinality`	`0`
`related_object_types`	`CosGraphs::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosReference::Relationship* interface.

RoleFactory attribute for ReferencedByRole	**value**
`role_type`	`CosReference::ReferencedByRole`
`maximum_cardinality`	`unbounded`
`minimum_cardinality`	`0`
`related_object_types`	`CosGraphs::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosRelationship::Relationship* interface.

9.6 References

1. James Rumbaugh, “Controlling Propagation of Operations using Attributes on Relations.” *OOPSLA 1988 Proceedings*, pg. 285-296.

2. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, “Object-oriented Modeling and Design.” Prentice Hall, 1991.