Relating Distributed Objects

Bruce E. Martin
SunSoft, Inc.
2550 Garcia Avenue
Mountain View, California 94043 USA
Bruce.E.Martin@eng.sun.com

R.G.G. Cattell
SunSoft, Inc.
2550 Garcia Avenue
Mountain View, California 94043 USA
Rick.Cattell@eng.sun.com

Abstract

Many relational and object-oriented database systems provide referential integrity and compound operations on related objects using relationship mechanisms. Distributed object systems are emerging to support applications that access objects across distributed, heterogeneous system boundaries. Because the fundamental assumptions of distributed, heterogeneous, federated computing systems differ from database systems, supporting object relationships in such an environment requires different approaches to the representation and manipulation of relationships than those traditionally used in database systems. This paper describes the Relationship Service for SunSoft's Distributed Object Environment (DOE). We describe the fundamental assumptions of distributed object systems and motivate our design in that context.

Keywords: relationships, object-oriented systems, complex objects, distributed computing.

1.0 Introduction

Relationships are a fundamental and useful data modeling construct in a wide variety of data and object management systems. Their importance to database systems was underscored in both the relational model and entity-relationship modeling extensions [3]. In this first section, we examine some basic properties of relationships. We then proceed to look at *distributed object systems* and the use and realization of relationships in that context.

1.1 Entity-Relationship Model

Relationships are associations between logical entities [10]. A system may automate the representation, maintenance, and use of relationships in a variety of ways:

- It may provide identifiers to use in creating relationships, e.g., primary keys in the relational model or object IDs in an object-oriented model, and it may allow bidirectional traversals as well as more complex queries of relationships based on the entities involved.
- It may provide a mechanism to define, examine, and modify relationships. This may include constraining the *roles* of a relationship according to the types of entities that may be related.
- It may provide *referential integrity* constraints. Referential integrity may be defined with varying levels of sophistication; for example, a system may simple avoid dangling references or it may manage the existence of objects based on relationships.
- It may support more advanced relationship constraints, e.g. to define the *cardinality* of relationship roles.
- The most sophisticated systems allow selective *propagation of operations* such as copy and delete through relationships between entities (sometimes called compound or composite objects).[9]

Some of the early relational DBMS products had little or no support for relationship constraints and referential integrity, but the industry has moved steadily in this direction with user demand. Object-oriented DBMS vendors have recognized the importance of good relationship support[1][2]. Relationships can also be useful in programming languages.

1.2 Relationships in Distributed Object Systems

The topic of this paper is relationships in *distributed object systems*. The mechanisms and implementation of

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

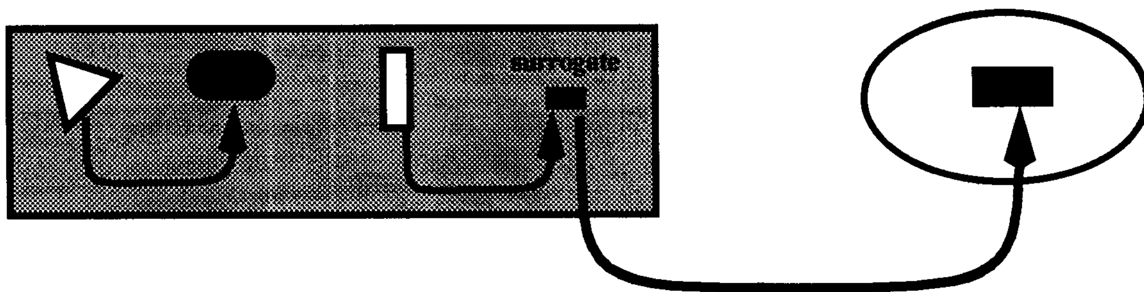

FIGURE 1. A distributed object system allows objects to request the services of other objects, possibly across heterogeneous system boundaries. On the left, an object requests the services of another object in the same system. On the right, an object requests the service of an object in another system by way of a surrogate object.

relationships in the context of distributed objects are quite different than in database systems. It is not possible to make the same assumptions as in the case of a monolithic database system from one vendor, where a data schema can be consulted for information about all the relationships one could happen upon, and the entities and relationships are always accessible.

We now proceed with an overview of distributed object systems. We then describe how SunSoft's Relationship Service provides relationships in a distributed object system. We will motivate our design based on the fundamentals of distributed object systems.

2.0 Distributed Object Systems

Distributed object systems[4][6] are emerging to support applications that access objects across distributed, possibly heterogeneous system boundaries. Such systems define a distributed object model that is mapped appropriately to native concepts in a wide variety of systems. Systems being bridged include database systems, file systems and persistent programming languages.

Distributed object systems embody the following principles:

- All entities are modeled as *objects*.
- *Interfaces*, not implementations, define objects.
- Distribution is *inherent*.
- There are no predefined universal *scopes*.
- Distributed object systems are *open*.

After discussing these principles in more detail, we describe and motivate our design for providing relationships in a distributed object system.

All entities are modeled as objects
In a distributed object system all entities are modeled as *objects*. Although the systems being bridged by a distributed object system may include entities that are not objects, they are not available across system boundaries unless they can be presented as objects.

Objects are accessed by clients using object references. Object references can be passed from one object to another. A client object holding an object reference for a target object can make a request for its services. The request may be local or it may cross heterogeneous system and administrative boundaries. It is the distributed object system's job to mask the differences from the client object.

Figure 1 illustrates a client object requesting the services of another object in the same system.The figure also illustrates a client object issuing a request on an object in a different system via a surrogate object. The surrogate object represents the remote object locally. The client object cannot distinguish the surrogate object from the remote object.

Interfaces, not implementations, define objects
In a distributed object system, objects are defined by their *interfaces*. An interface specifies a set of operations that defines the behavior of the object. The implementation of an object is separate and invisible; clients cannot depend on implementation properties, such as programming language, transient representation or persistent storage of the object. There can be multiple implementations of an interface. An interface does not imply any particular implementation(s), and a new implementation of any type of object may be added at any time.

This independence from implementation greatly simplifies object interaction, especially across heterogeneous system boundaries. (See [5], [11] for more discussion of separating interfaces from implementations.)

The independence from implementation also implies that there can be no distributed object system functionality that depends on implementation properties. In a database system, it is possible to provide "behind the scenes" functionality. For example, a database system can manipulate disk pages holding object state. In a distributed object system, there can be no such assumptions about the implementation of an object; all services must be expressed using interfaces.

Distribution is inherent

Distributed object systems are *federated* systems. Existing, disconnected heterogeneous systems are connected and made to interoperate. Gateways mask differences between systems by implementing mappings between concepts in each system, including interfaces, object models, object references and name spaces.

Distributed object systems have the potential of connecting large numbers of objects across system and administrative boundaries. There should be no limit to this; that is, the system should scale.

Federation and scalability lead to truly distributed system objects. Services that depend on a single, centrally administered repository of information are not acceptable. In particular, there is no authority, even a distributed one, that has information about all objects or even part of the information about all objects of one type. Instead, federated services are connected to other instances of the same service to widen their scope of discourse.

There are no predefined universal scopes.

In a distributed object system, it is not possible to get to all instances of a type; in contrast, you can find all the tuples in a relation in a DBMS. It is not possible to query the "known universe" — only explicitly maintained sets or other scopes may be queried. For the same reasons, there are no universal object identifiers: any identifier must be relative to some explicit scope.

Distributed object systems are open

Heterogeneous systems consist of components that typically come from a variety of suppliers. In order for the components to interoperate, the interfaces between the components need to be standard. Implementations of the components, however, need not be standard. Different suppliers can each provide implementations of a service supporting a standard interface.

Because the interfaces are standard, the services can be federated and interoperate. Customers can construct the best data management solution for their needs from various off-the-shelf interoperable components. Portions of that solution may be upgraded without "breaking" the rest of the system, because encapsulation separates implementations from interfaces. Customers may choose from multiple vendors for each component, and from each vendor they may choose an implementation "quality of service" for a component based on their needs, e.g. for speed versus guaranteed distributed data consistency.

2.1 The Object Management Group

The Object Management Group (OMG) is promoting standards for distributed object systems among system software vendors. The OMG has currently defined two sets of standards, known as CORBA and COSS. CORBA is the core communication mechanism which all OMG objects use: it enables objects to operate on each other. COSS provides standard services that support the integration of distributed objects.

CORBA

The Common Object Request Broker Architecture (CORBA) [7] defines an interface definition language (IDL) for objects. The language allows designers to specify interfaces as a set of operations and attributes. The language supports subtyping of interfaces. A function can be passed an object that supports a subtype of the interface expected by the function.

The CORBA defines object references. Object references are typed by interfaces specified in IDL. Object references unify access to objects. The client using the object cannot tell if the object being accessed is local or remote, who implements the object, or how it is implemented.

The CORBA also defines an interface repository. The interface repository contains descriptions of IDL interfaces. Such descriptions can be accessed at run time to implement type-safe interobject communication. Federating CORBA compliant systems requires correlating the interfaces in different interface repositories.

COSS

The Common Object Services Specifications (COSS) [8] defines a set of services for distributed object systems. The services are specified in OMG IDL and are intended to operate in CORBA environments. For flexibility, COSS defines functional components at a finer grain than complete DBMS functionality. All dependencies between components are explicitly defined as interfaces so that components from different sources interoperate.

Currently, COSS defines a name service for mapping human readable names to object references, a persistence service for persistently representing object state, an event service for decoupling communication between objects and an object life cycle service for creating, copying, moving and removing objects. Future specifications include transactions, concurrency control, externalization and object relationships.

3.0 The Relationship Service

The Relationship Service has been designed to provide the relationship functionality, useful for database applica-

tions, but given the principles of distributed object systems outlined in the previous section. The service is SunSoft's response to the OMG's request for relationship technology. Although the service is implemented in the context of the CORBA and COSS standards, the model we describe applies to other distributed object systems as well.

The Relationship Service supports the creation and manipulation of one-to-one, one-to-many and many-to-many binary relationships between typed objects. The Relationship Service enforces type, cardinality and referential integrity constraints on relationships. The relationships can be navigated and enumerated.

When objects are connected together using the Relationship Service, graphs of related objects are formed. The service supports compound operations on graphs.

3.1 Basic architecture of the Relationship Service

The Relationship Service supports the explicit representation of relationships between distributed objects. The service defines an object that supports the *Role* interface. A role *represents* an object in a relationship. Objects participating in relationships are called *related objects.*

Figure 2 illustrates the representation of the *containment* relationship between a document and a figure and a logo. The document, the figure and the logo are related objects. The document *contains* the figure and the logo. The figure and the logo are *contained in* the document. Containment is an example of a relationship. The Relationship Service is extensible; programmers can define other, application-specific, relationships.

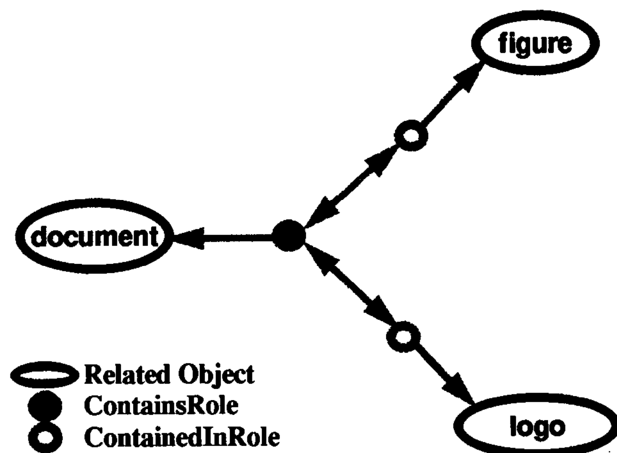

FIGURE 2. A document in a one-to-many containment relationship with a figure and a logo.

In Figure 2, the *ContainsRole* is an object that represents the document's role in containment. The *ContainedIn-Roles* are objects that represent the figure's role in containment and the logo's role in containment, respectively.

Since the related objects and the roles are *distributed* objects, there can be arbitrary distribution boundaries between them. As such, containment in our example is a logical concept, but not necessarily a physical concept. An object on one continent could contain an object on another. There are no implementation assumptions that the related objects are stored together.

Figure 3 gives the Role interface supported by all roles. The *set_related_object* operation establishes a role as a representative for an object in a relationship. The *get_related_object* operation returns that object.

```
interface Role {
      set_related_object(
         in RelatedObject obj)
         raises(TypeError,
         AlreadySet);
      get_related_object(
         out RelatedObject obj)
         raises(NotSet);
      create_relationship(
         in Role peer,
         out RelationshipId id)
         raises(TypeError,
         MaxCardinalityExceeded,
         NoRelatedObject);
      destroy_relationship(
         in RelationshipId id);
      get_relationships(out
         sequence<RelationshipId> ids);
      get_other_related_object(
         in RelationshipId id,
         out RelatedObject
         other_object);
      get_propagation_attribues(
         out PropagationAttributes pa);
```

FIGURE 3. The Role Interface

Connected roles represent binary relationships. The *create_relationship* operation creates a bidirectional connection between two roles and returns an identifier for the relationship. The *destroy_relationship* operation removes a binary relationship. The *get_relationships* operation enumerates all of the relationships in which the role participates.

The *get_other_related_object* operation navigates a relationship and returns an object reference for the other object in the relationship. In Figure 2, the get_other_related_object operation executed at the document's ContainsRole would return the figure, given the identifier for the relationship between the document and the figure; similarly, the get_other_related_object opera-

tion executed at the figure's ContainedInRole would return the document.

The *get_propagation_attributes* operation is discussed in detail in Section 3.3.2. Roles also support *link* and *unlink* operations that roles use to communicate with each other when relationships are created and destroyed.

Roles and Related Objects are typed by their IDL interfaces. For example, in Figure 2 the behavior of the document, figure and logo objects are defined by the Document, Figure and Logo interfaces. The roles of specific relationships are defined by subtypes of the Role interface. Thus, the ContainsRole and ContainedInRole interfaces are subtypes of the basic Role interface. Because they are subtypes, it is possible to pass them to functions that generically operate on relationships, that is, code that depends only on the Role interface.

Related objects that participate in multiple relationships are represented by multiple roles: one for each role the object plays in each relationship. Figure 4 illustrates this. Besides participating in the containment relationship, the document also participates in a reference relationship with the dictionary; the document references the dictionary and the dictionary is referenced by the document. The ReferencesRole represents the document's role in the reference relationship while the ReferencedByRole represents the dictionary's role.

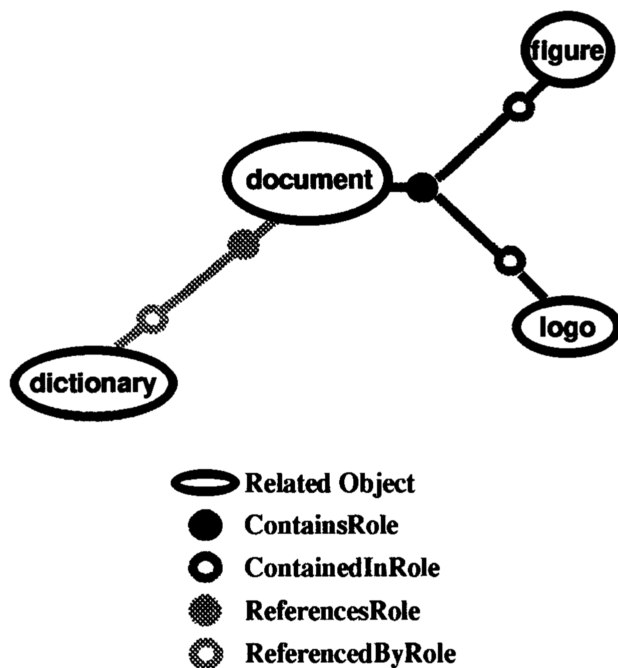

FIGURE 4. The document is also in a reference relationship with a dictionary.

In the entity-relationship model[3], the related objects are "entities" and the role objects are the roles. An instance of a relationship is represented by the identifier returned by a role when a connection is made. The relationship type is represented by the interfaces of the roles, as stored in the CORBA interface repository. Relationship attributes are represented as operations on roles, parameterized by the identifier.

3.1.1 Rationale

The basic architecture of the Relationship Service has several desirable properties for distributed object systems. Roles do not introduce performance or availability bottlenecks when navigating a relationship between distributed objects. The configuration of related objects and roles can be optimized appropriately for navigation or query functionality. Finally, the service is scalable.

We discuss each of these in more detail.

Performance and Availability

A role represents a related object in a relationship. An alternate approach introduces an object for each relationship, rather than roles. Our example of Figure 2 would instead be represented by the ContainmentRelationship objects of Figure 5.

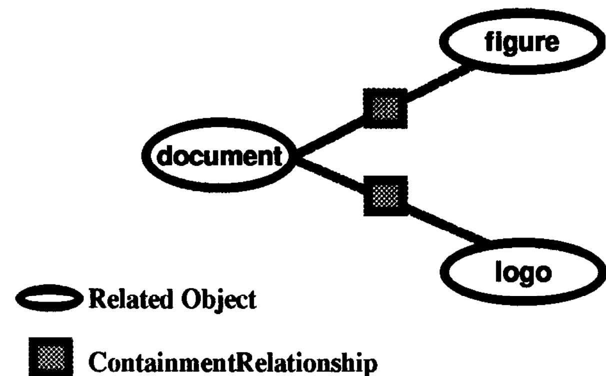

FIGURE 5. An alternate approach: relationship objects rather than roles.

This approach can introduce performance and availability bottlenecks when navigating a relationship between distributed objects. In a distributed object system, the cost of accessing a remote object is far greater than the cost of accessing a local object. Furthermore, remote objects can fail independently and thus be availability bottlenecks. In short, distribution boundaries are significant.

Consider the example. If the relationship object is isolated, that is it is not clustered with either the document or the figure, navigating the relationship from the document to the figure necessarily involves a remote request to the relationship object. Furthermore, the relationship object must be available. It effectively decreases the availability of navigating from the document to the figure because it introduces a new point of failure.

Rather than isolating the relationship object, we could cluster the relationship object with either the document or the figure. While this would improve the navigation performance in one direction, it would still require a remote request in the other direction.

We rejected the relationship object approach in favor of roles. When related objects are clustered with their roles, navigating a relationship between distributed objects does not introduce performance or availability bottlenecks. Figure 6 illustrates our example configured this way.

Navigating the relationship from the document to the figure does not require a remote request. Furthermore, since an related object and its roles are clustered, they fail together; the roles do not introduce availability bottlenecks.

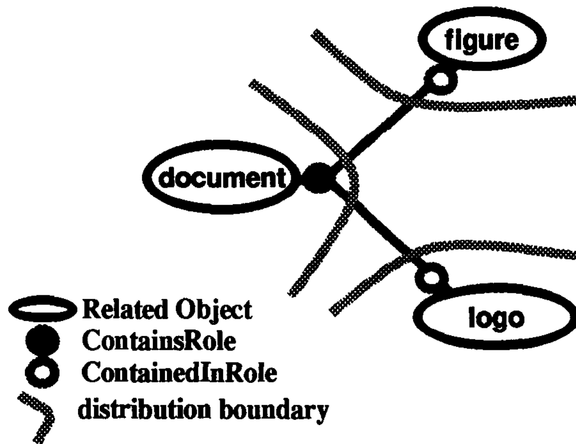

FIGURE 6. Clustering roles with the related objects

Flexible Configuration

Figure 6 illustrates a configuration of related objects and roles that is optimized for applications that *navigate* distributed objects. However, for applications that require efficient *querying* of relationships, clustering roles is a more appropriate configuration. Figure 7 illustrates the roles for several containment relationships clustered together.

The Relationship Service depends only on the Role *interface*. It does not depend on how the implementations of roles and related objects are clustered. Several configurations, including those of Figure 6 and Figure 7, are possible.

Scalable

Roles can be created independently, anywhere an instance of the Relationship Service exists. Since there is no single authority managing all roles, the service scales; creating a new role does not incur an additional distributed system-wide overhead. The Relationship Service is a *federated* service. Roles implemented by one instance of the service can be connected to roles implemented elsewhere.

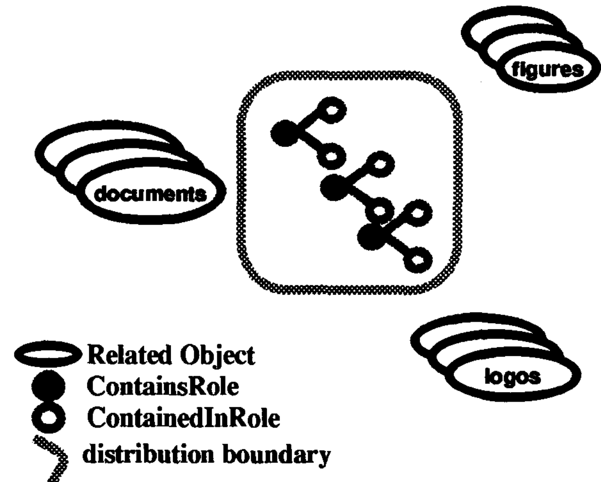

FIGURE 7. Clustering roles enables efficient querying of relationships

Because there is no single administrative authority, it is not possible to enumerate or query all of the relationships of the universe. As shown in Figure 7, however, it is possible to efficiently enumerate or query all of the relationships for some well-defined scope.

3.2 Relationship Constraints

The Relationship Service enforces several constraints on the relationships between objects. In particular, the service allows type, cardinality and referential integrity constraints to be expressed and enforced.

Type

Relationships are constrained by the types of the participants. For example, *employment* is a relationship between companies and people and not between fruit trees and frogs. Such type constraints are implemented using CORBA/IDL typing mechanisms.

Since the types of Roles and Related Objects are expressed as IDL interfaces, representations of the interfaces are stored in the interface repository. These type representations can be manipulated at run time. Using the type representations, the Role *create_relationship* operation enforces the type constraints of the relationship. If the type constraints are violated, the operation fails and raises a TypeError exception.

Cardinality

Relationships are also constrained by the number of objects that can participate in each role. For example, traditional marriage is a one-to-one relationship between people. Similarly, containment is a one-to-many relationship and employment is a many-to-many relationship.

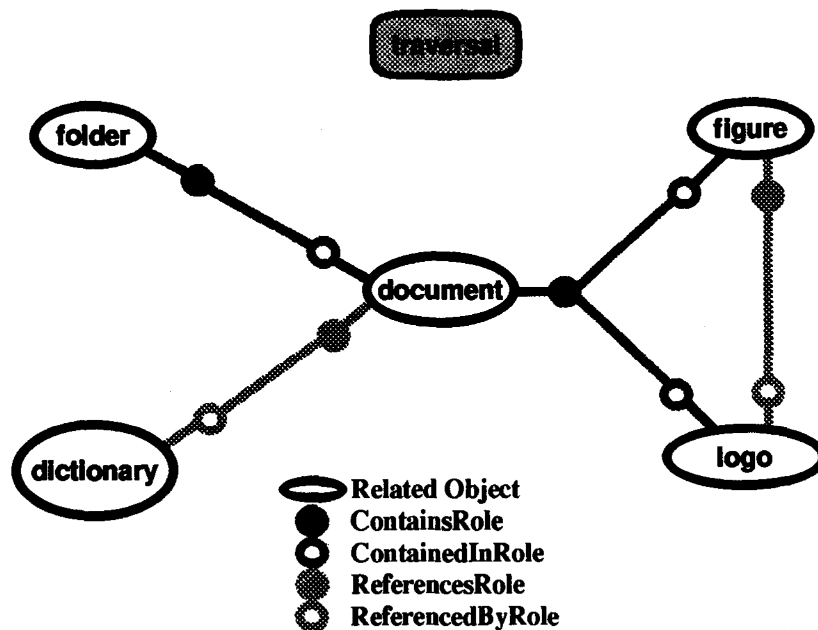

FIGURE 8. An example graph of related objects. The folder contains the document; the document references the dictionary and contains the figure and the logo; the figure references the logo.

As with type constraints, the Role *create_relationship* operation enforces the cardinality constraints of the relationship. If the cardinality constraints are violated, the operation fails and raises the MaxCardinalityExceeded exception.

Referential Integrity

When roles are connected, they form bidirectional connections that can be navigated in either direction. Referential integrity implies that object A is connected to object B, if and only if object B is connected to object A. If roles are connected under the control of transactions, referential integrity is guaranteed.

Existential integrity, object A exists if and only if object B exists, is supported as well and is explained in more detail in Section 3.3.2.

3.3 Graphs of Related Objects

When objects are related using the Relationship Service, graphs of related objects are created. After describing the basic components of a graph, we describe the Relationship Service's support for applying compound operations to a graph of related objects.

Graphs are defined by nodes and edges. The related objects are the nodes of the graph and the relationships are the edges of the graph. Traversal objects navigate graphs of related objects.

In Figure 8, the folder, the document, the dictionary, the figure and the logo are nodes. The containment relationships and the reference relationships are the typed edges of the graph. The folder *contains* the document; the document is *contained in* the folder. The document *contains* the figure; the figure is *contained in* the document. The document *contains* the logo and the logo is *contained in* the document. On the other hand, the document *references* the dictionary; the dictionary is *referenced by* the document. Finally, the figure *references* the logo; the logo is *referenced by* the figure.

3.3.1 Nodes

The Relationship Service defines a Node interface for related objects. In Figure 8, the folder, the document, the dictionary, the figure and the logo all support the Node interface. The Node interface enables a client to traverse a graph of related objects in a standard way. In particular the Node interface defines an operation and an attribute for related objects to reveal their roles. Figure 9 gives the IDL declaration for the Node interface.

Using the *roles_of_node* attribute, clients visit nodes, learn about the node's roles and navigate the relationships to visit adjacent nodes. The *roles_of_type* operation can be used to navigate particular types of relationships.

```
interface Node {
   readonly attribute
      sequence<Role> roles_of_node;
   Roles roles_of_type(
      in CORBA::InterfaceDef role_type)
};
```

FIGURE 9. The Node interface

3.3.2 Propagation Attributes

The roles in the graph have *propagation attributes*.[9] Propagation attributes define the rules for propagating operations from one node to another through the connected roles. A propagation attribute is a pair (operation, propagation value). The value is either *deep, shallow, none* or *inhibit.*

Deep means that the operation is also applied to the relationship and to the neighboring node.

Shallow means that the operation is also applied to the relationship, but not to the neighboring node.

None means that the operation has no effect on the relationship and no effect on the neighboring node.

Inhibit is meaningful for the delete operation. If a node has a role with a propagation attribute (delete, inhibit), the node cannot be deleted if the node is related to another. The relationship ensures the node's existence.

Figure 10 illustrates the propagation attributes for the containment relationship. Since the propagation attribute for copy, as defined by the ContainsRole, is *deep*, the copy operation is applied to the document and to the figure it contains. On the other hand, since the propagation attribute for delete, as defined by the ContainedInRole, is *shallow*, the delete operation is applied to the figure, to the relationship between the figure and the document but not to the document.

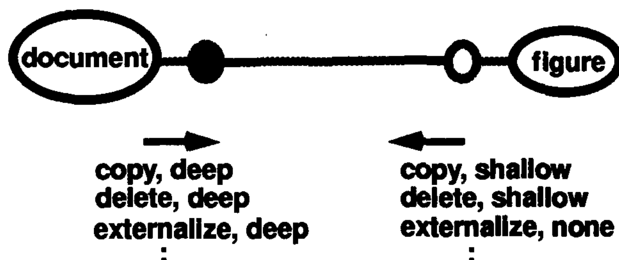

FIGURE 10. Propagation attributes for the containment relationship

Notice that propagation attributes are directed. The delete operation is deep when navigating from the ContainsRole to the ContainedInRole but is shallow when navigating in the other direction.

3.3.3 Traversal Object

The Relationship Service defines an object that supports the *Traversal* interface. The *Traversal* interface defines generic *traverse* operation that given a starting node and

an operation produces a finite set of nodes and roles affected by the operation. The traversal object uses the Node interface to determine a node's roles. Based on the propagation attributes of each role, the traversal determines the set of involved nodes and roles.

As the traversal object visits nodes and navigates roles, it may revisit a node due to cycles in the graph. The traversal object detects the cycles and represents them in the set of nodes and roles.

The traversal object does not actually apply an operation to the graph; the implementation of a compound operation can use the output of the generic traverse operation and apply the operation to the involved nodes and roles. Alternatively, a compound operation can be implemented directly by traversing the graph and applying the operation in a single pass.

3.3.4 Compound Operations

The Relationship Service defines a small set of compound operations that implement object life cycle operations. In particular, the service defines *copy, move, externalize* and *delete* operations. The service is extensible; application programmers can implement other compound operations on a graph of related objects.

An example

Copy is an example of a compound operation; it does not apply to a single object in the graph but to several objects, depending on the semantics of the relationships between the objects.

We apply the copy operation to the graph of related objects in Figure 8. The compound copy operation starts at the folder and proceeds as follows.The folder reveals its *ContainsRole*. The *ContainsRole* indicates that the propagation value for copy is *deep*. As such, the document also needs to be copied. The document reveals that it has three roles, a *ContainsRole* connected to the figure and the logo, a *ContainedInRole* connected to the folder and a *ReferencesRole* connected to the dictionary.

The *ReferencesRole* indicates that the propagation value for copy is *shallow*. As such, copy is not applied to the dictionary, it's roles are not considered and the new document will be connected to the old dictionary.

The *ContainsRole* connecting the document to the figure and the logo indicates that the propagation value for copy is *deep*. As such, the figure and the logo need to be copied. The figure reveals its *ReferencesRole* and its *ContainedIn*

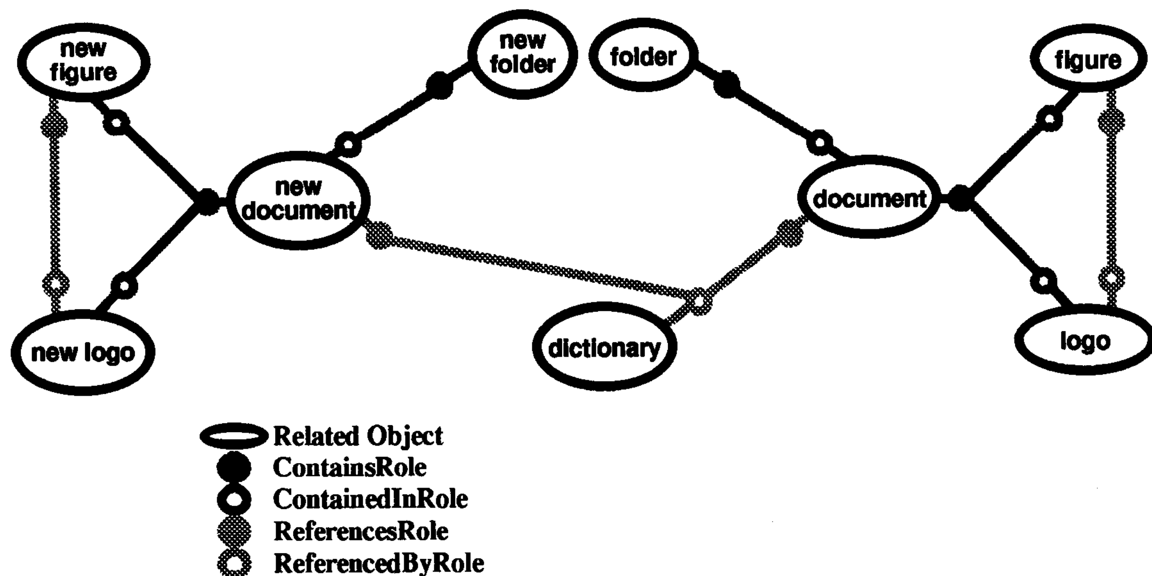

FIGURE 11. The result of applying copy to the graph, starting at the folder.

Role. The logo reveals its *ContainedInRole* and its *ReferencedByRole*.

When propagating a copy operation to a node that participates in relationships with different propagation semantics, it is possible that the propagation value for copy is *deep* by one relationship and *shallow* by another relationship. If a node is copied, then the shallow connections to it are *promoted* to deep.

This happens in several places in the example. The *ContainedInRoles* connecting the document to the folder, the figure to the document and the logo to the document are promoted since the figure and the document were copied. Similarly, the *ReferencesRole* between the figure and the logo is promoted because the logo is copied. The copy of the figure should not be connected to the old logo.

Figure 11 illustrates the result of applying copy to the graph, starting at the folder.

4.0 Experience

The Relationship Service was implemented using SunSoft's implementation of the CORBA specification. Applications that use the Relationship Service are typically designed using the Entity-Relationship Model; the relationships are naturally mapped to the Relationship Service. For example, a desktop of folders and documents was implemented using the containment and reference relationships illustrated here. Similarly, a distributed card trading game used the Relationship Service to relate trading cards and game players.

Application designers must decide when to use the Relationship Service and when to use lower-level CORBA object references. We have found that the Relationship Service is appropriate to use when an application needs to navigate connections between objects in both directions, needs to extend connections with attributes and operations, needs to allow third parties to manipulate connections or needs compound operations on graphs of related objects. Those capabilities are not available with CORBA object references.

Fortunately, relationships can be used heavily in distributed applications without a significant performance degradation over CORBA object references. When configuring objects with their roles, (see Figure 6) navigating a relationship between distributed objects has performance similar to navigating object references to distributed objects.

5.0 Conclusions

We have described SunSoft's Relationship Service and motivated its design, given the fundamentals of distributed object systems.

As discussed in Section 2.0, distributed object systems have different design goals than database systems. Distributed object systems are open, federated systems to support applications that access objects across distributed, heterogeneous system boundaries. All entities in a distributed object system are modeled as objects. *Interfaces*, not implementations, define objects.

In Section 3.0 we described the Relationship Service. The service supports binary, bidirectional, one-to-one, one-to-many, many-to-many relationships between objects. It enforces type, cardinality and referential integrity constraints on relationships between distributed objects. It

defines the Role interface which provides operations for creating, deleting, enumerating and navigating relationships. It supports flexible configuration of roles to optimize navigation or query functionality. It supports compound operations on graphs of related objects.

The Relationship Service is designed to work in a distributed object system. The service is defined by the Role, Node and Traversal interfaces. Any system that can support the interfaces can participate in the service. There are no assumptions about common implementation or storage of objects. The service is federated and scales. There is no single authority with knowledge of all relationships. Creating and manipulating a relationship does not have system-wide cost.

6.0 References

[1] R. G. G. Cattell, *Object Data Management*, Addison-Wesley, 1991, Revised 1994.

[2] R. G. G. Cattell, *The Object Database Standard: ODMG-93*, Morgan Kaufmann, 1993.

[3] P. P. Chen, "The Entity-Relationship Model: Towards a Unified View of Data," *ACM Transactions on Database Systems*, 1, 1, March 1976.

[4] Bruce E. Martin, Claus H. Pedersen and James Bedford-Roberts, "An Object-Based Taxonomy of Distributed Computing Systems." In *Readings in Distributed Computing Systems*, published by IEEE Computer Society Press. Also in *IEEE Computer* special issue on distributed computing systems, August, 1991.

[5] Bruce E. Martin, "The Separation of Interface and Implementation in C++." In *The Evolution of C++*, edited by Jim Waldo, published by MIT press. Also in *Proceedings of the 3rd USENIX C++ Conference*, April, 1991, Washington, D.C.

[6] John R. Nicol, C. Thomas Wilkes and Frank A. Manola, "Object Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, June, 1993.

[7] Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, December, 1991.

[8] Object Management Group, "The Common Object Services Specification, Volume 1", OMG Document Number 94.1.1, January, 1994.

[9] James Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations." *OOPSLA 1988 Proceedings*, pg. 285-296.

[10] James Rumbaugh, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[11] Alan Snyder. "Encapsulation and Inheritance in Object-oriented Programming Languages", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA). Association of Computing Machinery, 1986.