

Open GIS Consortium

35 Main Street, Suite 5
Wayland, MA 01778
Telephone: +1-508-655-5858
Facsimile: +1-508-655-2237

Editor:
Telephone: +1-703-830-6516
Facsimile: +1-703-830-7096
ckottman@opengis.org

The OpenGIS™ Abstract Specification **Topic 8: Relationships Between Features**

Version 4

OpenGIS™ Project Document Number 99-108r2.doc

Copyright © 1999, Open GIS Consortium, Inc.

NOTICE

The information contained in this document is subject to change without notice.

The material in this document details an Open GIS Consortium (OGC) specification in accordance with the license and notice set forth on this page. This document does not represent a commitment to implement any portion of this specification in any companies' products.

While the information in this publication is believed to be accurate, the Open GIS Consortium makes no warranty of any kind with regard to this material including but not limited to the implied warranties of merchantability and fitness for a particular purpose. The Open GIS Consortium shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

The Open GIS Consortium is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks, or other special designations to indicate compliance with these materials.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any means (graphic, electronic, or mechanical including photocopying, recording, taping, or information storage and retrieval systems) without the permission of the copyright owner. All copies of this document must include the copyright and other information contained on this page.

The copyright owner grants member companies of the OGC permission to make a limited number of copies of this document (up to fifty copies) for their internal use as a part of the OGC Technology Development process.

Revision History

Date	Description
26 March 1999	Carry forward 99-108r1; update for new document template following guidance of change proposal 99-010 (w/ friendly amendments) to remove Section 1 boilerplate from individual topic volumes, approved 9 February 1999; correct Figure 3-2 (FR_RelationshipAttributes).

This page is intentionally left blank.

Table of Contents

1. Introduction	1
1.1. The Abstract Specification.....	1
1.2. Introduction to Relationships Between Features.....	1
1.2.1. Examples.....	1
1.2.1.1. Scenario 1	1
1.2.1.2. Scenario 2	2
1.2.1.3. Scenario 3	2
1.2.2. Rationale.....	3
1.3. References for Section 1	3
2. The Essential Model for Relationships Between Features	4
2.1. Relationship Types.....	4
2.2. Degree	4
2.3. Roles.....	4
2.4. Role Types	4
2.4.1. Role Types and Feature Types	4
2.4.2. Roles Types and Cardinality.....	5
2.4.3. Role Types and Ordering.....	6
2.5. Directionality.....	7
2.6. Read-Only Relationships.....	7
2.7. Constraints and Integrity Maintenance.....	7
2.8. Lightweight and Heavyweight Relationships.....	7
2.9. Feature References	8
2.10. Faking Heavyweight Relationships	9
2.11. Writing generic software	10
2.12. Profiles.....	11
2.13. References for Section 2.....	11
3. Abstract Model for Relationships Between Features	12
3.1. FR_RelationshipType.....	12
3.2. FR_RoleType.....	12
3.3. FR_RelationshipAttributeType.....	12
3.4. FR_Role	12
3.5. FR_Relationship	12
3.6. FR_RelationshipAttribute	14
4. Appendix A. Additional Background.....	15
4.1. Analysis of binary relationships and integrity maintenance	15
4.1.1. Major Cardinality Types.....	15
4.1.2. Deletion Dependencies of Cardinality Types.....	15
4.1.2.1. List of Operations	15
4.1.2.2. List of Dependencies.....	16
4.2. Non-deletion Dependencies	17

<i>4.2.1. Schema Constraints</i>	17
4.3. References	17

1. Introduction

1.1. The Abstract Specification

The purpose of the Abstract Specification is to create and document a conceptual model sufficient enough to allow for the creation of Implementation Specifications. The Abstract Specification consists of two models derived from the Syntropy object analysis and design methodology [1].

The first and simpler model is called the Essential Model and its purpose is to establish the conceptual linkage of the software or system design to the real world. The Essential Model is a description of how the world works (or should work).

The second model, the meat of the Abstract Specification, is the Abstract Model that defines the eventual software system in an implementation neutral manner. The Abstract Model is a description of how software should work. The Abstract Model represents a compromise between the paradigms of the intended target implementation environments.

The Abstract Specification is organized into separate topic volumes in order to manage the complexity of the subject matter and to assist parallel development of work items by different Working Groups of the OGC Technical Committee. The topics are, in reality, dependent upon one another—each one begging to be written first. *Each topic must be read in the context of the entire Abstract Specification.*

The topic volumes are not all written at the same level of detail. Some are mature, and are the basis for Requests For Proposal (RFP). Others are immature, and require additional specification before RFPs can be issued. The level of maturity of a topic reflects the level of understanding and discussion occurring within the Technical Committee. Refer to the OGC Technical Committee Policies and Procedures [2] and Technology Development Process [3] documents for more information on the OGC OpenGIS™ standards development process.

Refer to Topic Volume 0: Abstract Specification Overview [4] for an introduction to all of the topic volumes comprising the Abstract Specification and for editorial guidance, rules and etiquette for authors (and readers) of OGC specifications.

1.2. Introduction to Relationships Between Features

Topic 5 of the Abstract Specification [5] introduces features, an abstraction of the entities in the real world. Entities in the real world do not exist in isolation. Typically an entity in the real world is related to other real-world entities in a variety of ways. This Topic introduces an abstraction for the relationships between entities in the real world. This abstraction is modeled as relationships between the features introduced in Topic 5.

1.2.1. Examples

1.2.1.1. Scenario 1

Consider the relationship between two real world entities, for example a road and river. The road may pass over the river on a bridge, pass under the river through a tunnel or even pass through the river at a ford. The digital representation of these two real world entities might be two *feature instances*, one of *feature type* ‘road’ and one of *feature type* ‘river’. Both ‘road’ and ‘river’ *feature types* might define that instances of that *feature type* have their real-world location represented as a linear geometry measured in a 2-dimensional spatial reference system. From this digital representation it may be possible to infer that the real world road and river intersect at some location, but it is not possible to determine which one passes under the other. (We make the simplifying assumption that the road and river features are broken up to the point that they will intersect with each other at most once).

This problem can be resolved without recourse to relationships. For example we could require ‘road’ and ‘river’ *feature types* to include an integer attribute (called ‘z-order’). The *feature* with the larger value for ‘z-order’ would be designed to cross over the other *feature*. However this relies on an interpretation of attributes that cannot be mechanically derived from the schema.

Now consider how the information might be modeled explicitly as a *relationship*. A new *relationship type* could be defined. We shall call this the ‘road-cross-river’ *relationship type*. Instances of this *relationship type* relate two *features*, one of *feature type* ‘road’ and one of *feature*

type 'river'. The *relationship type* defines two *role types*, one called 'road' and one called 'river'. Because there are only two roles, this is known as a binary *feature relationship*. The *feature type* 'road' is defined to include the 'road' *role type*. Thus features of the *feature type* 'road' may play the 'road' *role* of the *relationship*. The information about how the road crosses the river would be held as an attribute on the *relationship* itself, since it is not a information about either of the *features* in isolation.

As an alternative approach, consider the situation where the 'road' and 'river' *feature types* are both substitutable for the *feature type* 'linear feature' (see Topic 5 for an explanation of substitutable *feature types*). A new binary *relationship type* is defined with *role types* 'crosses' and 'crossed'. The *feature* that crosses is above, and the *feature* that is crossed is below. Both *role types* are defined on the *feature type* 'linear feature' or any *feature types* that are substitutable for it. Thus this *relationship type* can be used to model roads crossing rivers and rivers crossing roads, as well as roads crossing roads and perhaps (in some unusual circumstances) rivers crossing rivers. This *relationship* does not have an attribute defined on it, and consequently it cannot describe a road intersecting a river at the same height.

1.2.1.2. Scenario 2

A second scenario considers the relationship between a set of houses and the office of the agent that is trying to sell them. This might be represented as instances of *feature types* 'house' and 'office'. The 'house' and 'office' *feature types* might define that the real-world locations of the relevant entities be represented by a point geometry.

The problem can be resolved without recourse to relationships. For example we could require the 'office' *feature type* to include a string attribute called 'office-name' and the 'house' *feature type* to include a string attribute called 'selling-office-name'. The office *feature* whose 'office-name' attribute matched the 'selling-office-name' attribute of a house *feature* would be deemed to be the related office. However, this again relies on an interpretation of attributes that cannot be mechanically inferred from the schema.

The alternative is to model the relationship explicitly. Again this can be modeled with a binary *relationship*, where the two *roles* are 'house' and 'office'. However in this example an instance of the *relationship* may involve more than two *features*. An instance of this *relationship* always includes an office but it may involve arbitrarily many houses. This is a one-to-many *relationship type*. By contrast the *relationship types* described in Scenario 1 were one-to-one.

Consider the additional *feature type* 'ranch'. This is defined to be substitutable for the *feature type* 'house'. Then the *relationship type* defined above can be used to relate an office to a set of houses and ranches.

Defining a new *relationship type* implicitly defines certain types of associations between *feature types*. The 'road-cross-river' *relationship type* defined 'road' and 'river' *role types*. These *role types* were defined on certain *feature types* and constrained the types of feature that could participate in the *relationship*. Thus the 'road-cross-river' *relationship type* implicitly related the 'road' and 'river' *feature types*. This Topic does not deal with associations between *feature types* other than those which arise implicitly from the definition of *relationship types*.

In addition, it should be stressed that the *relationships* described in this Topic do not constitute the only way to model the associations that exist in the real world. Many of these can be better modeled in other ways. Consider a set of real world entities that are represented as a set of *features* where each *feature* has a geometric attribute with coordinates measured in the 'British National Grid' spatial reference system. One might say that *feature* A 'isSouthOf' *feature* B if the maximum y value of the geometry of *feature* A is less than the minimum y value of the geometry of *feature* B. Rather than explicitly establish a multitude of *relationships* to express this, one could employ a function defined on the *feature type*. This might be called 'isSouthOf' and return a boolean value that indicates whether the *feature* is indeed south of a second feature (passed as an argument to the method). This would calculate on-demand whether 'isSouthOf' is true between a pair of *features*.

1.2.1.3. Scenario 3

An example of a *relationship* with three *roles* might be the *relationship* between a bridge that allows a road to cross a river. We might wish to explicitly relate 'road', 'river' and 'bridge' features where they satisfy this constraint.

The classic example from the non-geographic world is of a person checking a book out of a library. A ternary *relationship* is used to relate the 'book', 'person' and 'library' features. This example

lends itself to a further development. The *relationship* might include an attribute ('checked-out-date') giving the date at which the book was checked out. The *relationship* essentially represents the event of the book being checked out. This arrangement allows the same book to be checked out by the same person from the same library more than once. The 'checked-out-date' attribute does not belong on any of the individual features. We could have included a list of checkout dates on the book feature itself (much as is done in the real world) but this does not allow us to retain the information about who checked the book out.

Geographic equivalents of this example are not as easy to come by. For example one might have thought that the road/river/bridge *relationship* would benefit from an attribute on the *relationship* giving the date when the bridge allowed traffic over the river. But due to its static nature, this could just as reasonably be an attribute on the bridge feature itself.

Including attributes on a *relationship* makes most sense when multiple *relationships* of the same *relationship type* can be used to relate the same set of features under different conditions. This suggests one example that relates a 'pontoon bridge', 'road' and 'river'. The *relationship* is used to indicate that a particular pontoon bridge was used to open a particular road to traffic over a particular river. Because such bridges can be moved when the work is done in one location, a pontoon bridge can participate in a number of such *relationships*. Indeed it could be used on more than one occasion to open the same road over the same river. Under such circumstances, it would be useful to include attributes on the *relationship* indicating when the bridge was in use.

1.2.2. Rationale

In attempting to model the real world, there is a need to model the associations that exist between entities in the real world. If no mechanism is provided, then these associations will be modeled as mysterious coincidences between attributes on *features*. Without additional information, it is not possible to mechanically interpret and manipulate these. By formalizing the concept of a *relationship*, this Topic attempts to provide the basis upon which specifications can make concrete suggestions.

In the spirit of the Abstract Specification, Topic 5 introduces the concepts of *feature* and *feature type* but does not seek to define particular *feature types*, let alone particular *features*. Similarly this Topic introduces the concepts of *relationship* and *relationship type*. But it does not dictate particular *relationship types*.

In a later section the concept of a *feature reference* is introduced and discarded. Essentially a feature reference is equivalent to the pointers familiar from various programming languages. A feature reference allows one object to point to another. This discussion of why these can be discarded in favor of relationships is necessarily delayed until we more fully understand what we mean by a relationship.

1.3. References for Section 1

- [1] Cook, Steve, and John Daniels, Designing Objects Systems: Object-Oriented Modeling with Syntropy, Prentice Hall, New York, 1994, xx + 389 pp.
- [2] Open GIS Consortium, 1997. OGC Technical Committee Policies and Procedures, Wayland, Massachusetts. Available via the WWW as <<http://www.opengis.org/techno/development.htm>>.
- [3] Open GIS Consortium, 1997. The OGC Technical Committee Technology Development Process, Wayland, Massachusetts. Available via the WWW as <<http://www.opengis.org/techno/development.htm>>.
- [4] Open GIS Consortium, 1999. Topic 0, Abstract Specification Overview, Wayland, Massachusetts. Available via the WWW as <<http://www.opengis.org/techno/specs.htm>>.
- [5] OpenGIS™ Abstract Specification, OpenGIS™ Project Documents 99-100 through 99-116, available through www as <<http://www.opengis.org/techno/specs.htm>>.

2. The Essential Model for Relationships Between Features

2.1. Relationship Types

Every *relationship* is an instance of a particular *relationship type*. The relationship type determines the various aspects of the relationship described below.

2.2. Degree

The relationship type defines the *degree* of the relationship. The degree is the number of different roles (see below) in a relationship. Relationships must be of degree 2 or more. Relationships of degree 2 are also referred to as *binary* relationships.

2.3. Roles

A relationship relates a number of features. Features participating in relationships may perform different and distinct functions or *roles* within a relationship. The relationship type is defined by *n roles types*, where *n* is the degree of the relationship. Thus a binary feature relationship type is defined by two role types.

For example one might wish to model the flow of rivers using a new, binary relationship type called 'river-flows'. The two roles might be called 'flows-from' and 'flows-into'. Clearly it is necessary to distinguish between the two roles in this example since river A flowing into river B is very different from river B flowing into river A. Although multiple role types are required to define a feature relationship type, there is no requirement for the role types to be different. Thus a symmetric, binary feature relationship type like 'adjacency' might be defined by two uses of the same role type.

2.4. Role Types

The role type is a specification of what roles a feature has and how relationships connect to those roles. A role type is only defined on particular feature types and thus constrains the features that can have that role. The role type also constrains how many relationships can be reached from a role and whether the ordering of those relationships is important. This Topic enumerates the minimum set of characteristics one would expect to be considered when defining a role type:

- Feature types
- Cardinality
- Ordering

2.4.1. Role Types and Feature Types

Feature types are defined to have zero or more role types and thus define which relationships a feature can participate in. To continue the 'river-flows' example, both 'flows-from' and 'flows-into' roles are defined on the 'river' feature type. Thus only features of feature type 'river' or some feature type that is substitutable for 'river' can participate in the 'river-flows' relationship. Thus it is not possible to establish an instance of the 'river-flows' relationship type between two roads.

A role type can be defined on more than one feature type. Thus the 'flows-into' role type might be defined on feature-type 'sea' in addition to feature type 'river'. Thus rivers can flow into the sea. However it is not possible for the sea to flow into a river, since the 'flows-from' role type has not been defined on the 'sea' feature type.

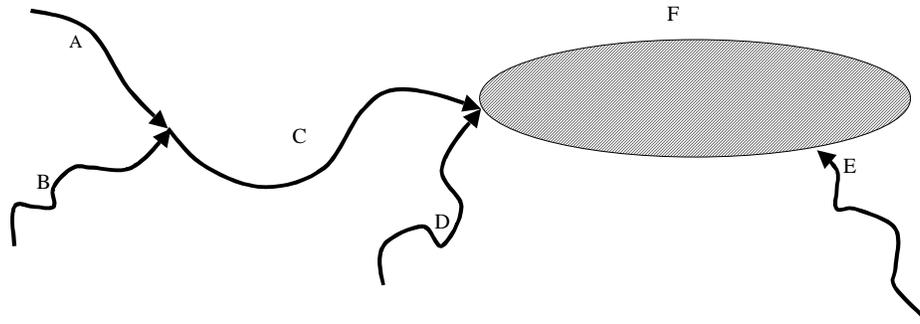


Figure 2-1. Rivers and Sea Roles and Relationships

In the example above (Figure 2-1), F is the sea and the rivers A,B,C,D and E run into it and each other. There are five instances of the ‘river-flows’ relationship; A->C, B->C, C->F, D->F, E->F. For example river A ‘flows-into’ river C whereas river C ‘flows-from’ rivers A and B. The figure below (Figure 2-2) shows another way of looking at the same example. It explicitly displays the relationships but does not show the feature geometries. Note that the river features have two roles, whereas the sea feature has one.

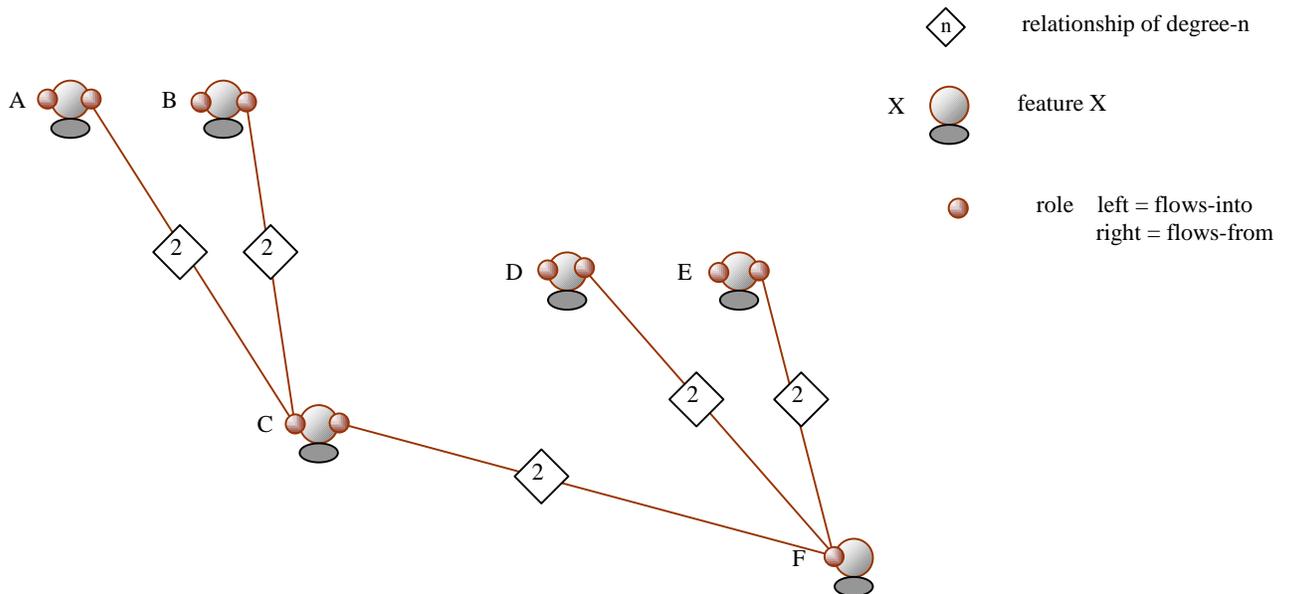


Figure 2-2. River and Sea Roles and Relationship Graph

2.4.2. Roles Types and Cardinality

The role type places a constraint on the number of relationships that can use that role. This is referred to as its *cardinality*, or sometimes its *multiplicity*. Cardinality is expressed as a subset of the non-negative integers. Common values are one; zero-or-one; zero-or-more; and one-or-more.

In the ‘river-flows’ relationship type the ‘flows-into’ role has a cardinality of zero-or-more, whereas the cardinality of the ‘flows-from’ role is zero-or-one. (We assume that rivers do not split). In the diagram the ‘flows-into’ role on feature F has three relationships, whereas on features A,B,D and E it has none. The ‘flows-from’ role has a single relationship on all the river features. Since we assume all rivers ultimately flow into the sea, we could have set the cardinality of the ‘flows-from’ role type to be precisely one.

Binary relationship types that have role types whose cardinalities are of the form ‘zero-to-one’ or ‘zero-to-many’ are often characterized with phrases such as one-to-one, many-to-one or many-to-many. We might describe the ‘river-flows’ relationship type as ‘many-to-one’.

2.4.3. Role Types and Ordering

If the role type permits more than one relationship, then the role type must indicate if the *ordering* of relationships is meaningful.

Consider the example of a relationship type that allows ‘bus-route’ features to be described as a list of ‘road-segment’ features. This can be modeled with a many-to-many, binary relationship type. A particular ‘road-segment’ may be part of multiple ‘bus-routes’ and a ‘bus-route’ can be made up of multiple ‘road-segments’. However to determine the actual path the bus is intended to take over time requires additional information about the order in which the ‘road-segments’ are to be traversed.

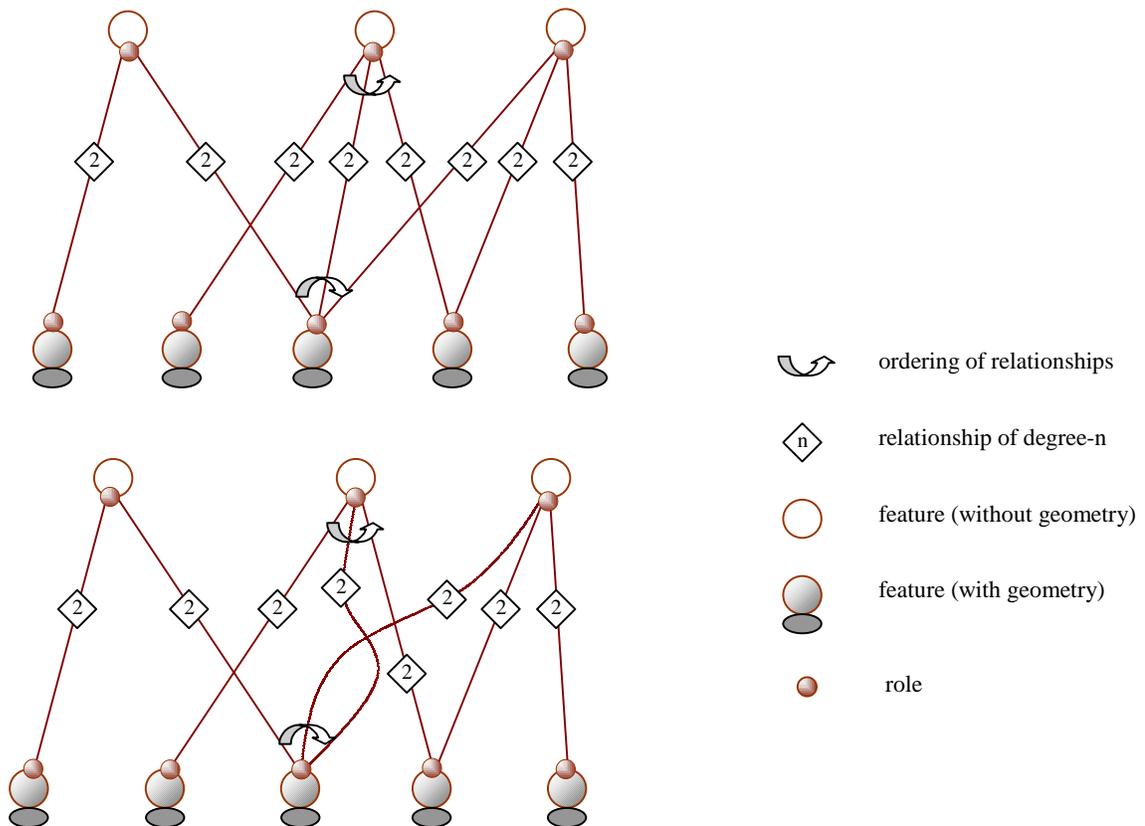


Figure 2-3. Bus Routes and Road Segments Example

The diagram above (Figure 2-3) uses an arrow to indicate the ordering of relationships that are included in a role. Clearly it is possible for all the role types in a relationship type to indicate that ordering is important. The ability to specify the importance of ordering on a per role-type basis is one of the reasons that ordering should not be regarded as an attribute of the relationship itself.

Continuing with the bus-route example, one could imagine situations where the same road-segment was traversed more than once, in the same direction, by the same bus-route. In this case not only does the feature participate in more than one feature relationship of the same feature relationship type, but it does so related to the same ‘other’ feature. In the absence of any ordering we would have two feature relationships of the same feature relationship type relating the same pair of features.

2.5. Directionality

Relationships are *multidirectional*. When features are related using a relationship, the relationship can in principle be navigated from any role to any other role. Some directions may be more efficient to navigate in than others; but that is an implementation issue and thus not part of this abstract specification.

2.6. Read-Only Relationships

Relationship types allow us to define an increasingly sophisticated abstraction of the real world. How these relationships are calculated or managed persistently is not part of the definition. In many cases relationships provide a way of encoding information about the real world that is not easily represented in any other way. However there are also many examples of relationships that could, in principle, be derived from other information. The abstract specification should not proscribe either of these approaches. However, if a simple relationship is derived from complex information, it is not possible to update the model to establish the relationship without information extra to that retrievable from the relationship. For these reasons, it may be useful to be able to indicate that a relationship type is *read-only*.

2.7. Constraints and Integrity Maintenance

From the above discussion, it is clear that the definition of a relationship type includes constraints. These are assumed to exist in addition to constraints defined elsewhere. For example the concept is included in Topic 5 where attributes could be defined to have constraints; for example a real attribute called 'length' might be constrained to disallow negative values.

The problem with relationships is that multiple features are involved, and the consequences of not having any transactional control become more obvious.

Consider an example based on a simple network topology. Let us have feature types 'node' and 'link'. Nodes have a geometric attribute that is a point geometry and links have a geometric attribute that is a linear geometry. However we can model their connectivity without relying on their associated geometry. Consider a binary relationship type called 'network' (this Topic does not require that relationship types can be identified by name, but it proves convenient for the purposes of description). The relationship itself has a Boolean attribute called 'isStart'. It also has roles types 'node' and 'link'. The 'node' role has a cardinality of 0/N, whereas the 'link' role has a cardinality of 2. That is every link feature has a role of type 'link' that contains two 'network' relationships. By comparison a node feature has a role 'node' that might have zero or more 'network' relationships.

Without any transactional support, we assume that features must be created one at a time. Using the 'network' feature relationship requires that the start and end nodes of a link must be present before the link can be created. As the constraints become more complex, the correct ordering of feature creation becomes increasingly difficult.

Indeed it can rapidly become impossible. Consider an example with two feature types 'A' and 'B' and the binary relationship type called 'difficult'. This has two roles 'A' and 'B', both with a cardinality of 1. That is a feature of one feature type can only exist without encountering a constraint violation when related to a feature of the other feature type. Clearly it is impossible to create features of these types one at a time.

The Simple Features Specifications place the responsibility for integrity maintenance on the server. In the pursuit of interoperable clients and servers, it is manifestly unreasonable to expect all clients to be well behaved and the responsibility for integrity maintenance lies with the server.

It should be noted that relationships introduce their own integrity constraints beyond those introduced by the relationship type. Specifically it should not be possible to navigate a relationship to a non-existent feature.

2.8. Lightweight and Heavyweight Relationships

The terms *lightweight* and *heavyweight* are introduced to describe two extreme categories of relationship types. Lightweight relationships have the simplest requirements and heavyweight relationships have the most complex requirements. The terms are introduced to facilitate discussion and do not represent favored combinations of requirements.

Lightweight and heavyweight relationships differ in three areas:

1. **Existence:** heavyweight relationships are treated as first-class objects and have an existence (and consequently an identity) of their own, independent of the features that they relate. Some or all of the roles in a heavyweight relationship may be 'empty', that is there is no feature participating in that role. Lightweight relationships have no independent existence and only exist as long as none of their roles are empty.
2. **Attributes:** heavyweight relationships may have attributes associated with them, lightweight relationships cannot. If a relationship has attributes, one would expect to be able to query to find those relationships with certain attributes. In turn this would require an ability to identify the relationships satisfying the query.
3. **Degree:** heavyweight relationships can have a degree of two or more, lightweight relationships have a degree of two.

Figure 3-1 describes heavyweight relationships. To make it describe lightweight relationships it is necessary to set the cardinalities of the **FR_RelationshipType.has role type** and **FR_RelationshipType.has role** associations to 2, and that of **FR_RelationshipType.has attribute type** association to zero.

Having identified a relationship, it is meaningful to ask what features participate in its roles. It should be noted that, from the perspective of a relationship, each role has, at most, one feature participating in it. This is only a practical concern for heavyweight relationships since only they can be identified independently of any features.

The existence of lightweight relationships can only be inferred through participating features. Given a feature and a role, it is possible to identify the related feature(s). If the role permits multiple relationships, it is not possible to identify the individual relationships explicitly.

If one considers a binary, many-to-many, lightweight relationship type, then it is possible to have two feature relationships that are indistinguishable.

The diagram below (Figure 2-4) shows a binary, one-to-many, heavyweight relationship. The relationship must be heavyweight because the relationships themselves have attributes. These attributes belong to the individual relationships; there are three of these in the diagram below. Each of these relationships relates two features because it is a binary relationship.

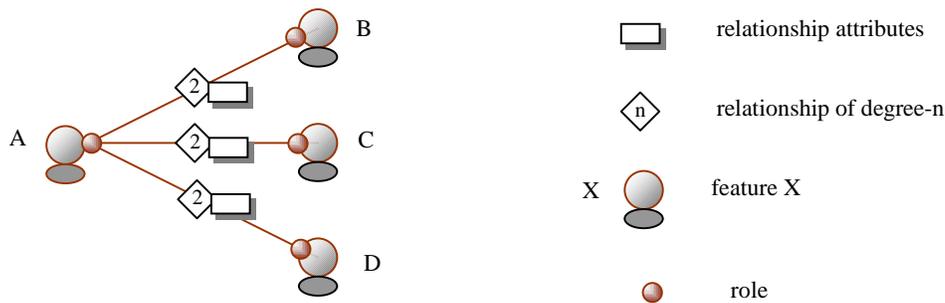


Figure 2-4. A Binary, One-to-Many, Heavyweight Relationship

The phrase 'one-to-many' relationship is something of a misnomer. It is true that, viewed from feature A with the relationship type specified, it is possible to navigate to features B, C and D. But that does not mean that we are dealing with a single relationship instance (we have three relationship *instances* between A and B, C and D). This distinction becomes clearer when we consider putting attributes on the relationships themselves.

2.9. Feature References

The concept of a 'reference' is familiar from many systems, and can be extended to include *feature references*. Feature references are essentially lightweight, binary, many-to-one relationships that

can only be navigated in the ‘to-one’ direction. In their simplest form, relationships can be used to represent feature references, but they retain the ability to be navigated in both directions.

Feature references are analogous to the ‘pointer’ or ‘object reference’ familiar from many programming languages. A feature reference allows one feature to identify another feature. The related issue of *feature identity* is dealt with in more detail in Topic 5. This Topic examines those elements that make relationships special. It suggests that, because feature references can be so closely modeled by relationships, there is no need for a separate analysis of feature references.

However the concept of a feature reference makes it clear that in many circumstances one should be able to exploit relationships without having to use all the machinery associated with them. Specifically, although relationships involve all the features equally, it is possible to ‘view’ a set of relationships from a specific feature. For example traversing a ‘to-one’ relationship from a specified feature need be no different from getting the value of an attribute whose type just happens to be another feature (the discussion of whether the return type is a feature or a feature-identifier is left to Topic 5). Indeed this can be extended to include the update of lightweight relationships provided sensible default behaviors are included.

2.10. Faking Heavyweight Relationships

It is possible to mimic elements of relationships using just features and their attributes (this was described in Section 1.2). However these approaches rely on some unwritten rules to make the connections between features. For example one needs to know that a certain attribute from one feature type must be matched against another attribute from another feature type. One might invent a naming convention for attributes that makes this explicit. In such a case the rules that govern the naming convention are the missing element. Indeed this provides one way of specifying how relationships should be modeled. However it is not possible, in principle, to add relationships without introducing something new.

This Topic has introduced a number of elements, some more advanced than others. For example we can have *lightweight, binary relationships* and we can have *heavyweight, degree-n, attributed relationships*. It is, however, possible to model the latter with the former.

Consider an attributed relationship of degree-3. We introduce a new feature type, ‘myRelation’, to hold the same attributes as we would like to hold on the relationship. This feature type does not include a geometric attribute. We then define three lightweight, binary relationships. For each of these three relationships there is a role that has cardinality one and only allows instances of the feature type ‘myRelation’ to participate. The role name can be generated from the role name from the corresponding ternary relationship and the name of the new feature type. The other role for each of the three relationships has the same characteristics as one of the roles in the original heavyweight relationship.

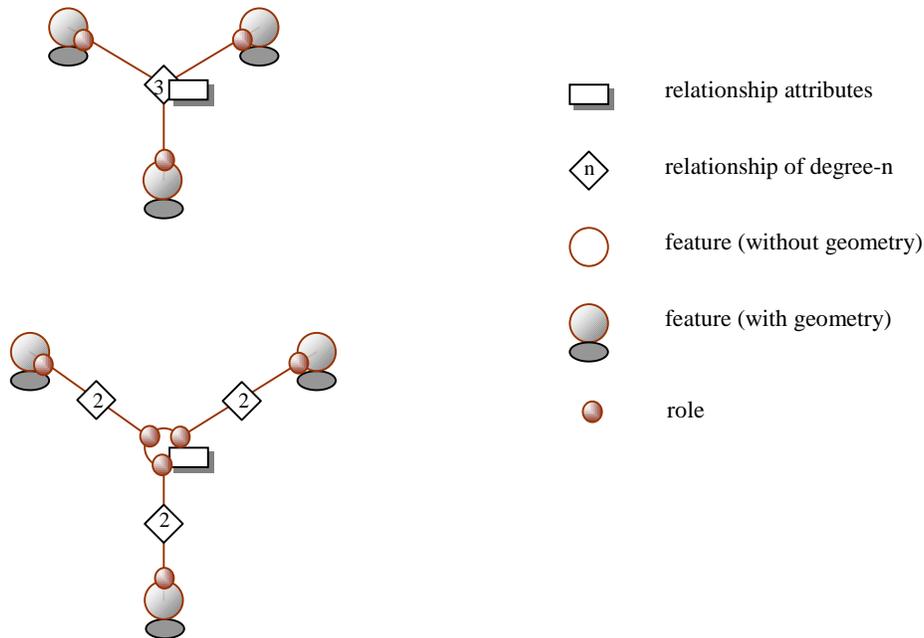


Figure 2-5. Heavyweight and Faked Heavyweight Relationships

The disadvantage of this approach is that we have had to create one new feature type, three binary relationships and six roles. A heavyweight relationship would have required the definition of one ternary relationship and three roles. That said it should be noted that the extra definitions can be created in a mechanical manner. The more important point is that there is no need for any additional information external to the system. This is not the same as trying to fabricate relationships using attributes.

2.11. Writing generic software

Everywhere a constraint is based on a feature type, a feature of any substitutable feature type is designed to satisfy the constraint. This is essentially the definition of substitutable for feature types. One of the benefits of substitutable feature types is that it allows generic software to be written, provided it only exploits aspects of the nominated feature type.

Consider a feature type called 'net-link' and the feature types 'gas-pipe' and 'water-main' that are substitutable for it. A binary relationship type 'net-flow' is defined with roles 'flows-into' (cardinality zero-to-many) and 'flows-from' (cardinality zero-or-one). Both roles are defined on features with a feature type substitutable for 'net-link'. (We shall ignore the problem that most networks can be reconfigured to alter the direction of flow through the use of valves and the like.)

Using the 'net-flow' relationship type, it is possible to build a model of both gas and water networks. In addition it is possible to imagine a generic network analyzer that follows the 'flows-from' role to trace a route through the network. If the analyzer is pointed to a gas-pipe it will analyze the gas network, if it is started on a water-pipe it will analyze the water network.

Unfortunately the 'net-flow' relationship type does not prevent one building a network that includes both gas-pipes and water-mains. It would be possible to define separate relationship types for the gas and water networks that would provide tighter constraints. But this has the disadvantage that not only considerably more elements need to be defined in the schema, but that software seeking to use it needs to be aware of this variety of virtually identical relationships.

It is not possible to use the additional constraints imposed by relationship types to both ensure a sensible network and allow the writing of generic software. In general one needs a programming language to express constraints. For example the constraint that pipes should become narrower in the direction of flow cannot be captured using the constraints described here. The discussion of how it might be possible to describe additional arbitrary constraints is left for another Topic.

2.12. Profiles

This Topic introduces the concept of a relationship and various support concepts, such as role, relationship type and role type. It does not prescribe the set of relationship types that should be implemented. This should be allowed to vary and it should be possible to discover such information from the schema. However there is frequently merit in agreeing on common elements used by a variety of schemata. This essentially means defining a *profile*. This Topic does not specify profiles, but we consider how it might apply to the generic problem of containment.

A common example of a relationship is the 'containment' relationship. For example consider an instance of the feature type 'hospital'. It might contain an instance of the feature type 'administrator'. Features that are contained by another feature do not have an existence independent of that feature. This distinction between features that have an independent existence and those that are part of another feature is common to many entity and object design methods [2, 3]. This means that the *only* relationship the administrator can take part in is the 'containeer' role in a relationship with the hospital. Furthermore the cardinality of the 'containeer' role is one. Any relationship type that has these characteristics could be considered to be modeling a containment relationship.

2.13. References for Section 2

- [1] OpenGIS™ Abstract Specification, OpenGIS™ Project Documents 99-100 through 99-116, available through [www as <http://www.opengis.org/techno/specs.htm>](http://www.opengis.org/techno/specs.htm).
- [2] Hull R. and King R., Semantic Database Modeling, Survey, Applications and Research Issues, ACM Computing Surveys 19 (3) 1987, 201-260.
- [3] Unified Modeling Language (UML)
<http://www.rational.com/uml/resources.html>

3. Abstract Model for Relationships Between Features

Relationships between features are described by the FR_Relationship package. This refers to the AT_Attribute and FT_Feature packages described in Topic 5. An overview of the FR_Relationship package is given in Figure 3-1.

3.1. FR_RelationshipType

All relationships have a type described by a relationship type. The relationship type has a string attribute called **name** that can be used to identify the relationship type. The complete specification of a relationship type refers to the definition of two or more role types. A relationship type has a derived integer attribute called **degree** that is the number of these role types.

3.2. FR_RoleType

Role types help define relationship types. Each role type helps to define precisely one relationship type. The role type must also be defined as one of the roles on at least one feature type (FT_FeatureType). The role type has a string attribute called **name** that can be used to identify the role type. It has an attribute called **cardinality** that is expressed as a set of integers, or as a range of integers where the upper limit can be left unspecified. It has a Boolean attribute called **ordered**.

3.3. FR_RelationshipAttributeType

Relationship attribute types help define relationship types. A relationship may be attributed, thus a relationship type can refer to zero or more relationship attribute types. A relationship attribute type helps define precisely one relationship type. A relationship attribute type is a specialization of a generic attribute type (AT_AttributeType). It inherits a string attribute called **name**, used to identify the attribute type, and an attribute called **type**. The type attribute has a domain that defines the type of attributes that can be held on a relationship.

3.4. FR_Role

The information that can be held on a feature is determined by the associated feature type. Just as a feature type determines the set of feature attributes on a feature, it determines the set of roles on a feature. The number of roles on a feature is the same as the number of role types defined on the associated feature type. Each role is an instance of an associated role type.

3.5. FR_Relationship

Each relationship is an instance of an associated relationship type. A relationship may be associated with zero or more roles, where the maximum number of roles is determined by the number of role types that define the associated relationship type (namely the **degree** attribute of the associated relationship type).

The number of relationships that can be associated with a role is determined by **cardinality** attribute of the associated role type.

If more than one relationship can be associated with a role, the ordering of the relationships at the role is retained if the **ordered** attribute of the associated role type is true.

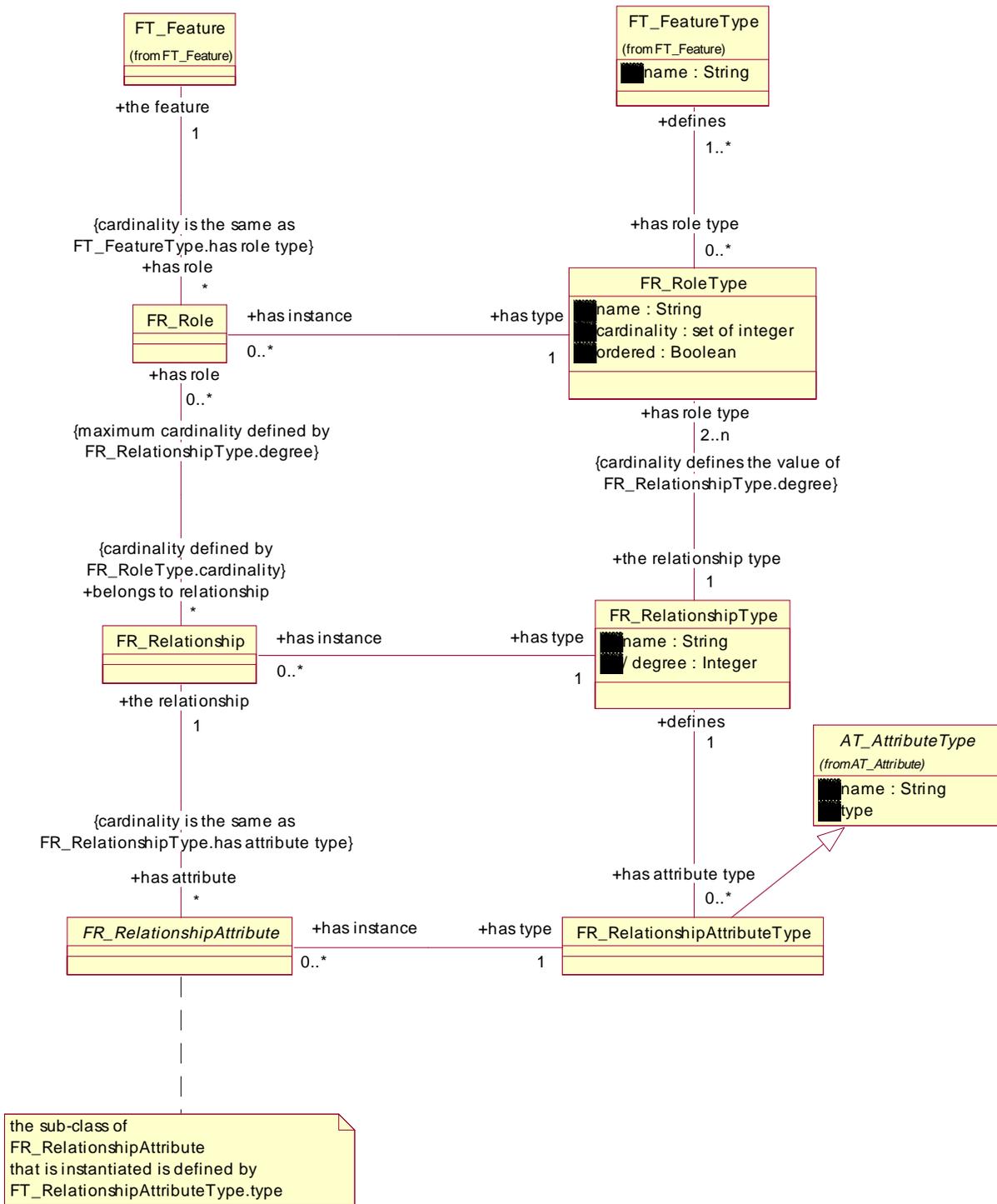


Figure 3-1. The FR_Relationship Package.

3.6. FR_RelationshipAttribute

A relationship may have a number of associated relationship attributes. Each relationship attribute is an instance of a relationship attribute type. The number and type of these relationship attributes is determined by the relationship attribute types defined on the relationship type.

The FR_RelationshipAttribute class is an abstract super-class. Relationship attributes are instances of classes that inherit from FR_RelationshipAttribute and AT_Attribute (see Topic 5). Inheritance from FR_RelationshipAttribute ensures that a relationship attribute is associated with a relationship and has a relationship attribute type. Inheritance from AT_Attribute ensures that it has an attribute called **value**.

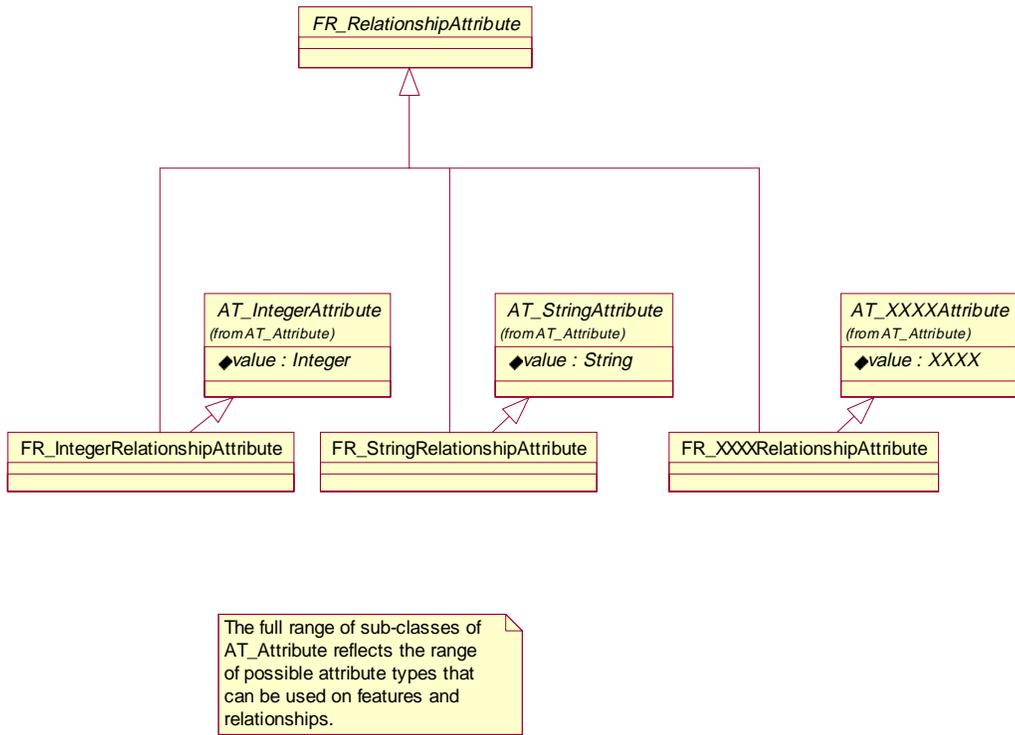


Figure 3-2. FR_RelationshipAttribute classes.

Figure 3-2 shows classes derived from FR_RelationshipAttribute that can be instantiated. Compare with the diagram for FT_FeatureAttribute in Topic 5.

However the type of the **value** attribute for a relationship attribute is determined by the **type** attribute of the associated relationship attribute type. For this reason there are a number of 'typed' relationship attribute classes. The UML diagram above does not seek to show a complete set of these classes and uses FT_XXXXRelationshipAttribute as an example of a relationship attribute capable of holding a value of type XXXX.

4. Appendix A. Additional Background

4.1. Analysis of binary relationships and integrity maintenance

This analysis goes into a greater level of detail, specifically for binary relationships. This is not meant to imply that such a treatment is required.

What this does is provide a way of specifying how to maintain a level of integrity without the need to introduce a transactional mechanism. It is, of course, an open question as to whether one would prefer to see transactional support or a complex mechanism for describing how relationships are manipulated.

4.1.1. Major Cardinality Types

Ehlmann and Riccardi [1, 2] define a notation (not shown in this document) to denote cardinalities and bindings (constraints). A cardinality-relationship is defined as:

cardinality -to- cardinality

Cardinalities are:

1	
0/1	(i.e., 0 or 1)
M	(i.e., 1 or more)
0/M	(i.e., 0 or more)
<i>integer ..</i>	(e.g. 2.. means 2 or more)
<i>integer .. integer</i>	(e.g. 1..9 means 1 to 9 inclusive)

Given these four possible cardinalities and assuming binary relationships, ten distinct and basic feature relationship types can be constructed. Six of the sixteen possible combinations are duplicates since binary relationships are bi-directional (or at least, the directionality can be specified separately):

1. 1-to-1	e.g. state to capital city
2. 1-to-M	e.g. houses to housing subdivision
3. 1-to-0/1	
4. 1-to-0/M	e.g. cables to cables
5. M-to-M	e.g. telephone poles to aerial telephone cables
6. M-to-0/1	
7. M-to-0/M	e.g. bus routes to streets
8. 0/1-to-0/1	
9. 0/1-to-0/M	e.g. trees to parks
10. 0/M-to-0/M	e.g. rivers to parks

4.1.2. Deletion Dependencies of Cardinality Types

The existence of a relationship may imply a set of *constraints* on the lifecycle and attribute values of participating *features*: i.e. the state of a *feature* may be *dependent* on the state of another *feature*. Outside the context of a transaction all such constraints remain true if the system is in a consistent state.

Since current OpenGIS™ Simple *Feature* Specification [3] make no provision for transactions, we do not specify these deletion dependencies here.

4.1.2.1. List of Operations

The cardinality of a relationship places some basic constraints, but is insufficient on its own to determine behavior when a *feature* or relationship is deleted. For example, given two *features* in a

1-to-1 relationship, there are three possibilities regarding the deletion of one of the *features*. Either it is not allowed, or both the *features* and the relationship are deleted, or one of the *features* and the relationship are deleted. Bindings define the constraints on deletion of either *features* or relationships.

The following operations can alter relationships between *features*:

- adding a feature
- deleting a feature
- creating a relationship between two *features*
- destroying a relationship between two *features*
- changing a relationship for a feature so that it relates to different *features*, or exchanging one *feature* in a relationship with another *feature*

4.1.2.2. List of Dependencies

Explicit bindings deal with what happens when relationships are deleted explicitly (e.g. what happens to the related *features*?). *Implicit bindings* deal with implicit deletion of relationships when related *features* are deleted.

Ehlmann and Riccardi define a relationship as a cardinality-relationship plus bindings.

binding < cardinality-relationship > *binding*

They have defined a notation for *binding*. A list of the various binding types follows, with the notation for each type specified within parentheses after the name. Assume that a feature of class C1 has relationship R with *feature* of class C2:

Default Implicit (none): On delete of *feature* of class C1, existing R relationships are implicitly deleted provided the deletion does not violate the cardinality of C1 or C2. For example, if the relationship between C1 and C2 is 0/1-to-M, then when C1 is deleted, the relationship is also deleted. But if the relationship between C1 and C2 is 1-to-M, then the delete fails since deleting C1 violates the cardinality.

Example: A bus route is related to street segments <0/1-to-M>. If a bus route is deleted, then its relationships with street segments are deleted, but the street segments remain. If a street segment is deleted, the bus route remains unless it is the last remaining street segment for the bus route. In that case, that street segment cannot be deleted.

Propagate Implicit (|~): On delete of *feature* C1, existing R relationships are implicitly deleted. When the cardinality of C1 is violated by the delete of R, then the related C2 *feature* is deleted. In this way, the delete is propagated. If the delete of C2 fails, due to other relationship cardinalities or other constraints, then the delete of C1 and R is prevented. For example, if the C1 to C2 relationship is 1-to-M, then deleting C1 will delete the R relationship and also all *features* C2 related to C. If the relationship is 0/1-to-M, then the C2's are not deleted since they can exist without a feature C1.

For example, if the relationship between a county *feature* and land parcel *features* is <1-to-M>|~, then every county must have at least one land parcel, every land parcel must belong to exactly one county. The |~ binding means that the deletion of the last land parcel of a county would cause the county to be deleted as well. However, a county cannot be deleted explicitly since that would violate the 1-to-M relationship with land parcels. If the relationship were |~<1-to-M>|~, this would mean that a county could be deleted explicitly and that the deletion of the county would result in the deletion of all the related land parcels.

Minus Implicit (|-): On delete of *feature* of class C1, existing R relationships are never implicitly destructible. The relationships must be deleted explicitly first.

Default Explicit (none): An R relationship can be deleted explicitly provided that it does not violate the cardinality of C1.

For example, a <1-to-1> or <0/1-to-1> relationship cannot be deleted explicitly. However, a <0/1-to-0/1> relationship can be deleted.

Minus Explicit (X-): An R relationship cannot be deleted explicitly, only implicitly when related *features* are deleted.

Propagate explicit (X~): An R relationship is always explicitly destructible. The related *features* are implicitly deleted when their cardinality is violated.

If the relationship in the county to land parcel example were <1-to-M>X~, then: If a relationship is deleted between a county and a single land parcel, the land parcel is deleted.

The county is deleted only if its last land parcel is deleted. If a county is related to only one land parcel, then deleting the relationship will delete both the county and the land parcel.

Prime (*): The prime binding for a class C1 in a relationship R denotes that C1 is the *prime class* and the other class, C2, is the subordinate class. On delete of *feature* C1, existing R relationships are implicitly deleted and implicit deletes are done on all subordinate *features*. Also, when an R relationship is explicitly deleted, an implicit delete is done on the subordinate *feature*. Failure of an implicit delete on the subordinate *feature* causes the failure of the C1 *feature* delete or explicit R relationship delete if and only if the implicit delete is needed to maintain the cardinality of C1 in R. This binding can be given to only one class in a relationship.

Consider the bus route to street segment example, except with the relationship '<0/M-to-M>'. If a bus route were deleted, then all street segments related to that bus route but not related to any other bus routes would be deleted.

4.2. Non-deletion Dependencies

Deletion dependencies are not the only, or even the most useful, type of dependency to have that uses relationships between *features*. Various forms of integrity checking, validation, pre- and post-update checks, pre-instance-creation checks etc. are known (from existing commercial systems) to be vitally useful to large data repositories with complex updating requirements. Therefore, it would be premature to suggest specification of deletion dependencies without also attempting to specify a more complete set of dependencies which may be no harder to implement.

4.2.1. Schema Constraints

If the feature relationship type defines a 1:1 relationship between two feature types, this only applies the constraint if a relationship instance exists. However, some repository schemas may wish to assert a 1:1 existence constraint *even in the absence of any relationship* instance. Such a constraint is not part of this feature relationships abstract specification.

4.3. References

- [1] Object Data Management Group, 1997. The Object Database Standard: ODMG 2.0. San Francisco: Morgan Kaufmann Publishers Inc. ISBN 1-55860-463-4.
- [2] Ehlmann, B.K. and Riccardi, G., A. A Notation for Describing Aggregate Relationships in an Object-Oriented Data Model, 1994 International Conference on the Applications of Databases, Vadstena, Sweden, July, 1994. Also available at: <http://www.cs.fsu.edu/~riccardi/papers.html>
- [3] OpenGIS™ Consortium, 1997. OpenGIS® Implementation Specifications, Wayland, Massachusetts. <http://www.opengis.org/techno/specs.htm#implementation>