Journal of Systems Integration, 3, 133-161 (1993)
© 1993 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Structural Integration: Concepts and Case Study

J. GELLER*
geller@hertz.njit.edu
Y. PERL*
perl@vienna.njit.edu
Institute for Integ. Systems, CIS Department, and Center for Manufacturing Systems, New Jersey Institute of Technology, Newark, NJ 07102

P. CANNATA AND A. SHETH
amit@ctt.bellcore.com
Bellcore, 444 Hoes Lane, Piscataway, NJ 08854

E. NEUHOLD
neuhold@darmstadt.gmd.de
GMD-IPSI, Dolivostr. 15, D-6100 Darmstadt, Germany

(Received March 12, 1992; Revised February 1, 1993)

Abstract. When integrating the views of a large telecommunications application database at Bellcore, it was found that some pairs of view objects had significant structural similarities but differed semantically.[+] This observation motivated the design of the structural integration methodology, described in this article. Currently existing view and schema integration methodologies are based on semantic considerations. They allow integration only if two objects agree in their semantic and structural aspects. Structural integration permits the integration of objects even if they differ semantically. This article introduces structural integration for the case of full structural correspondence. We further develop an important special case, namely structural integration for classes with attribute partial correspondence. We use a subschema of the telecommunications application to demonstrate the applicability of structural integration to situations involving the complexities of real-world databases and applications. Algorithms for checking full structural correspondence of classes and databases are presented. Structural integration has several advantages, including the identification of shared common structures that are important for sharing of data and methods.

Key Words: Database and schema integration, structural integration, reusability, dual model, ER model, object-oriented model, structure and semantics.

1. Introduction

In the process of creating a database for a large application, views are defined that describe subsets of the data. Each view supports the data requirements of a group of users or a logically independent subset of applications. Once the different views have been created, one needs to integrate them into a single schema that describes all the data used by the application. This is called view integration. The process of creating a schema of a database shared by multiple applications also involves view integration. The process of integrating

*This research has been supported partially by grants from the Center for Manufacturing Systems at New Jersey Institute of Technology and from Bellcore.

[+]No implication may be made about whether Bellcore develops or supports this particular application or the corresponding database.

schemas from different databases (which may already exist) is called schema integration. The techniques applied to integrating views and schemas are quite similar [2, 25]. Many methodologies and techniques have been suggested for view and schema integration—Batini et al. [2] compare 12 of them; several others are discussed by Sheth and Larson [25].

Based on the distinction between structural and semantic representations in a data model presented below, all earlier view and schema integration techniques can be classified as semantic. These techniques determine what can be integrated based on the semantics of the application, but the integration is possible only if structural aspects match as well. In this article, we introduce a technique called *structural integration*, and provide algorithms for applying it. We also discuss its use with a large realistic application, and point out its advantages. As we will see, structural integration complements semantic integration. Different models assign different meanings to the terms *structure* and *semantics*.[1] A formal definition of our distinction appears in Section 3.1 and is further explored in [10].

Generalization is a useful technique for integrating view/schema objects [5]. Typically, when two entities are similar but not identical, i.e., they are identical in some properties and different in others, one can create a superentity that contains the common properties of the original entities. The properties in which the two original entities differ are described by two new subentities of the superentity which inherit its properties. The two new subentities, together with the superentity, contain the same information as the original two entities (see Figure 1). By creating the superentity, we save space in the description of the common properties which are now specified only in the superentity rather than in each of the two original entities. The process of generalization described above is common to both the extended entity-relationship (ER) models that support generalization and to many object-oriented models. For object-oriented models an additional advantage is *code reusability*, which is achieved by describing the common methods once in the superentity rather than twice in the original entities. The generalization process can be applied recursively, creating a hierarchy of entities. Properties of a superentity are inherited by its subentities through all levels of the hierarchy.

In the current methodologies, integration by generalization can be used when one can identify two entities that are similar in both structure and semantics. While studying the schemas of a large telecommunications application database, it was discovered that several subschemas had the same or very similar structures, but different semantics. That is, there were pairs of corresponding classes with comparable numbers and types of attributes and relationships, but no intuitively correct common superclasses for the pairs.

Unfortunately, integration by generalization cannot be applied to semantically dissimilar classes, even if structural similarities "invite" it. This problem exists for extended ER models and for the known object-oriented models. The desire to integrate subschemas of the telecommunications database which were similar in structure but different in semantics motivated the development of *structural integration*.

Structural similarities can only be exploited for integration if the data model supports a clear distinction between structure and semantics. The object-oriented Dual Model [20, 21, 22] is just such a model. Structural integration does not replace integration by generalization, but supplements it where the latter is not applicable.

Structural integration provides the following advantages. It allows sharing of the specification of properties including methods. Sharing of definitions of an object type by several

$$\boxed{\text{entity}_1} \quad \boxed{\text{entity}_2}$$

$$\begin{array}{rcl}
\text{property}_1 & = & \text{property}_1 \\
\text{property}_2 & = & \text{property}_2 \\
 & \vdots & \\
\text{property}_j & = & \text{property}_j \\
\text{property}_{j+1} & \neq & \text{property}_{j+1} \\
 & \vdots & \\
\text{property}_n & \neq & \text{property}_n \\
\text{property}_{n+1} & & \\
\text{property}_{n+2} & & \\
 & \vdots & \\
\text{property}_m & &
\end{array}$$

$$\boxed{\text{super-entity}}\ (\text{property}_1 \ldots \text{property}_j)$$

$$\begin{array}{rcl}
\boxed{\text{sub-entity}_1}\ \text{property}_{j+1} & \neq & \text{property}_{j+1}\ \boxed{\text{sub-entity}_2} \\
 & \vdots & \\
\text{property}_n & \neq & \text{property}_n \\
\text{property}_{n+1} & & \\
\text{property}_{n+2} & & \\
 & \vdots & \\
\text{property}_m & &
\end{array}$$

Figure 1. Generalizing two entities with common properties.

object classes results in a savings of specification (as with IS-A inheritance). This savings would be impossible in a methodology that does not support the distinction between structure and semantics.

Another advantage is cognitive in nature. Structural integration results in a structural schema that is smaller than the union of the classes of the two or more schemas that it integrates. This makes it easier for a human to get a quick understanding of the database by first studying the integrated (structural) schema and then applying this understanding to the two or more original schemas.

The Dual Model and structural integration can be said to support and exploit a novel form of *semantic relativism*. Semantic relativism was defined as "the ability to view and manipulate data in the way most appropriate for the viewer" [3]. Two forms of semantic relativism discussed in the literature are: (a) The ability to interpret a data model structure differently (e.g., a relation can be viewed as an entity or a relationship) [3]; and (b) the definition of multiple views (external schemas) over a database schema (conceptual schema or federated schema) to support different users' needs for viewing and using data differently [3, 23]. The Dual Model supports a novel form of semantic relativism where structural aspects are represented as types and semantic aspects are represented as classes. By mapping multiple classes onto a single type, multiple semantics (uses and meanings of data) are supported by a single structure. Structural integration allows us to exploit this form of semantic relativism in the context of integrating multiple views or schemas.

While many techniques relevant to view and schema integration have been proposed (e.g., see [24]), we are unaware of a published document that discusses the validity and usability of the proposed techniques by applying them to complex and large views and schemas found in real industrial applications and databases. We believe that it is important to test our work in complex real-world situations. This was done in the context of a large telecommunications application. In this paper, the structural integration is demonstrated using portions of the schema for this application's database, rather than using several simpler pedantic examples.

This article is the first in a two-part series. It introduces the notion of structural integration as well as the actual application for which structural integration was developed. The theory of structural integration is advanced in this article to the level that is necessary for the application. In a follow-up article, the theory of structural integration is further extended to deal with more difficult and general integration problems [11]. A short version of this article appeared as [9].

The article is organized as follows. In Section 2, we present an extended ER schema used to describe a telecommunications application database. In this description we identify two pairs of subschemas that are similar in their structures but are semantically different. It is necessary to represent structure and semantics separtely to afford structural integration. The object-oriented Dual Model [22] that supports this separation is described in Section 3. In Section 4 we present the Dual Model semantic representation of the telecommunications application schema described in Section 2. In Section 5, we present algorithms for performing structural integration. We define *full structural correspondence* and *attribute partial correspondence* for subschemas and discuss structural integration for these two cases. Section 6 presents the conclusions.

2. An Entity-Relationship Schema of a Telecommunications Application Database

In this section we discuss the ER schema of a telecommunications application database. We show structurally similar subschemas that cannot be integrated because of different semantics. This provides the motivation for the introduction of structural integration.

The ER model is popular in the telecommunications industry. Many application databases are described in various versions of the ER model. A particular extended ER model supports two abstractions. One is *subtypeof* or *IS-A*, graphically represented using a triangle labeled S and connected by a bold line to a parent entity type. The second is *roleof*, graphically represented with a triangle labeled R.[2]

Figure 2 shows a portion of the schema of a large telecommunications application database which is related to the SWITCH system of Bellcore.[3] For simplicity, attributes are not shown; however, as discussed in Section 5, they are considered when deciding structural

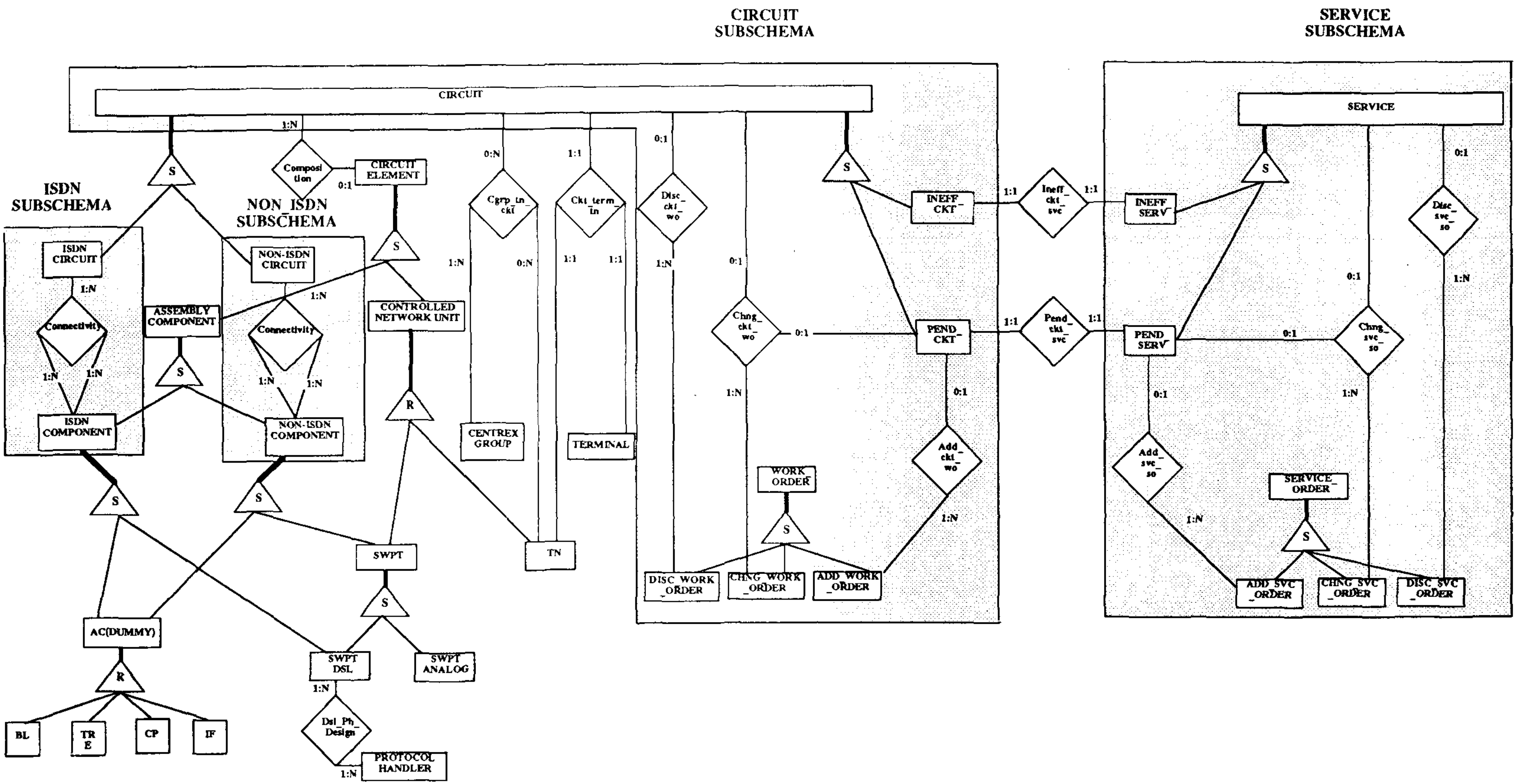

Figure 2. ER schema of the telecommunications application.

correspondences between schema objects. The figure contains two pairs of structurally similar subschemas (shown by four shaded regions in Figure 2): (*SERVICE*, *CIRCUIT*) and (*ISDN*, *NON_ISDN*).

The *SERVICE* subschema describes a request (by a customer) for a service (or a set of services) and the creation of a corresponding work order for the required service(s). A SERVICE_ORDER entity can be one of the following kinds, which are represented as its subentities: ADD_SVC_ORDER, DISC_SVC_ORDER and CHNG_SVC_ORDER. Another entity is SERVICE, describing a service offered by a telephone company to customers. A service can either be *in effect* or *pending*. Typically when a service is installed, it is marked pending, and the customer is told when it will become effective. These services are represented as subentities of SERVICE, namely INEFF_SERV and PEND_SERV. The relationship Add_svc_so connects the ADD_SVC_ORDER to the new PEND_SERV (pending service) created by the order. The relationship Disc_svc_so connects the DISC_SVC_ORDER to SERVICE. DISC_SVC_ORDER represents an order to discontinue a set (possibly singleton) of services that are either in effect or pending. The relationship Chng_svc_so connects the CHNG_SRV_ORDER to SERVICE. CHNG_SRV_ORDER represents an order to change a set (possibly singleton) of current services, either in effect or pending, to a new set of pending services.

The *CIRCUIT* subschema describes connections among work orders and circuits. These connections are similar to those in the *SERVICE* subschema. The difference between the two subschemas is that *CIRCUIT* has more "external" connections than *SERVICE*. The *CIRCUIT* and *SERVICE* subschemas have similar structures while their semantics are different. The structural similarity can be seen in the duplication of relationships and abstractions between pairs of corresponding entities. The sets of attributes of corresponding entities will in general be similar but not equal. The *ISDN* and *NON_ISDN* subschemas are structurally similar as well.

In view of the structural similarity of these two subschemas, one would like to integrate them. However, the only available tool in the extended ER model is generalization which can be applied only where similarity exists in both structure and semantics. The same problem occurs in all existing object-oriented database systems. The dual model overcomes this difficulty.

3. The Object-Oriented Dual Model

3.1. Separation of the Structure and Semantics of a Class

The Dual Model has been introduced in a number of previous publications [10, 20, 21, 22] and has been contrasted with other object-oriented approaches such as [7, 13, 18, 19]. Based on the Dual Model theory, the VML (Vodak Modeling Language) object-oriented database system has been developed [17]. In this paper we will limit ourselves to an informal review of the features that are absolutely necessary for understanding structural integration.

The Dual Model is an object-oriented model which uses *attributes*, *relationships*, *methods*, and *generic relations* to describe classes. Due to the widely differing terminology in the field, we find it necessary to state our use of these terms to establish common ground with the reader.

An attribute specifies printable values and does not relate to any other class. A relationship describes a user-defined connection to another class. A method is a program segment attached to the class. A generic relation is a commonly used (system-defined) connection from one class to another. All specialization relations are generic relations. A Dual Model schema consists of two levels, a structural schema and a semantic schema. Both these schemas make use of attributes, relationships, methods, and generic relations.

In Section 5.2, we will discuss the connections that exist between a structural schema and its corresponding semantic schema. In this section we will concentrate on the distinctions between these two schemas. To stress these distinctions, the building blocks of the structural schema are called *object types* and the building blocks of the semantic schema are called *object classes* (or just *classes*).

The following two definintions which were introduced in [10] capture formally our understanding of structure and semantics. The terms *aspect* will be used for attributes, relationships, methods, generic relations, and constraints.

Definition 1. As aspect of a specification is considered *structural* if either (1) It is composed of names, types, and logical or arithmetic operations; or (2) it is decidable whether this aspect is consistent with the mathematical representation of the class(es) it connects to.

Note that the name of a property is considered semantic in other models (e.g., ER model) but is not considered semantic in the Dual Model.

Definition 2. An aspect of a specification is considered *semantic* if either (1) It refers to actual instances of objects in the application; or (2) it is not decidable just based on the mathematical representation of the class(es) it connects to, whether this aspect describes properly the connection between the corresponding real-world objects and their features.

The relationships defined in an *object type* refer only to other object types, i.e., they stay strictly in the structural schema. The same applies to generic relations. The relationships defined in an *object class* refer only to other object classes, i.e., they stay strictly in the semantic schema. The same applies, again, to generic relations. Similarly, a method definition has both structural and semantic components, which are associated with an object type and an object class, respectively.

It is important not to confuse the notion of *object type* with *data type*. Data types can also be called attribute types. Typical examples of data types are INTEGER and REAL. Their values are stored directly with the object where they are defined. Relationships and generic relations refer to object types (or classes) but not to data types. Object types may contain relationships, which is not true for data types. Object types may be organized in a subtype hierarchy, which is also not true for data types.

Every object class must have a single associated object type, but one object type may have several associated object classes. Every instance in the database is an instance of one object class, and is therefore indirectly an instance of one object type. The object type contributes a description of the structural features of the instance. For example, it contributes the data types of the attributes.

The object class contributes different items of additional semantic information to an instance. It might establish that an instance of a class is a specialization of another class, *whether or not those two classes look similar in their properties*. For example, an object class `student` might be asserted to be a specialization of an object class `person`, even if the properties of `student` are not a superset of the properties of `person`.

One advantage of object-oriented databases over the ER model is that a class may have methods as properties. In the Dual Model, a method can be either a computational method or a path method. A *path method* is a chain composed of relationships and generic relations. It is used to retrieve an item of information I that is relevant to an instance of a class *a*, but I is stored with an instance of a class *b*. Typically, there is no direct connection between *a* and *b*, i.e., the shortest connection from *a* to *b* includes several other classes. The path method is then a path through the schema that starts at *a* and ends at *b*. A path method may be terminated by an attribute. A linear form of the graphical representation of a path method is now given. The term *Connection* stands for a relationship or a generic relation.

$$class_1 \overset{Connection_1}{\rightarrow} class_2 \overset{Connection_2}{\rightarrow} \ldots \overset{Connection_{n-1}}{\rightarrow} class_n \overset{Attribute}{\rightarrow} result$$

In the path from $class_1$ to $class_n$, the connection $Connection_k$ is said to be at position k. For a more complete definition of path methods see [22].

3.2. Graphical Description of Dual Model Schemas

The graphical description of a Dual Model schema consists of two figures, one providing the structural representation, and the other the semantic representation. In the following description, the building blocks of our graphical language [14] are introduced.

- A *class* or an *object type* is represented by a rectangle. Class names are written in lowercase letters. Object type names are written in capital letters. (There is no confusion possible, since classes and object types do not appear in the same diagram.)
- A composite class, such as a *set class* or *tuple class*, is represented as a rectangle with special borders. A set class is a class of set objects. A set class and a set object type are described using a bold line rectangle. A tuple class is a class of tuple objects. Both are denoted as a double-line rectangle.
- A specialization relation between classes (i.e., *roleof* or *categoryof*) or object types (i.e., *subtypeof*) is represented as a bold arrow. These terms will be defined in Section 3.3.
- The *setof* and *memberof* relations between a class (or object type) and its set class (or set object type) are represented by drawing the corresponding rectangles touching at one corner.
- *Relationships* are drawn as labeled arrows. A relationship name is written with its first letter as capital.
- Attributes and methods are not shown in this article.

Figure 3 shows the semantic aspects of the telecommunications schema in the Dual Model. Figure 4 shows both the semantic *CIRCUIT* and *SERVICE* subschemas, and the structural *OFFERING* subschema that corresponds to both. Figure 5 shows the semantic *ISDN* subschema and the semantic *NON_ISDN* subschema and their corresponding structural subschema *I-CIRCUIT*.

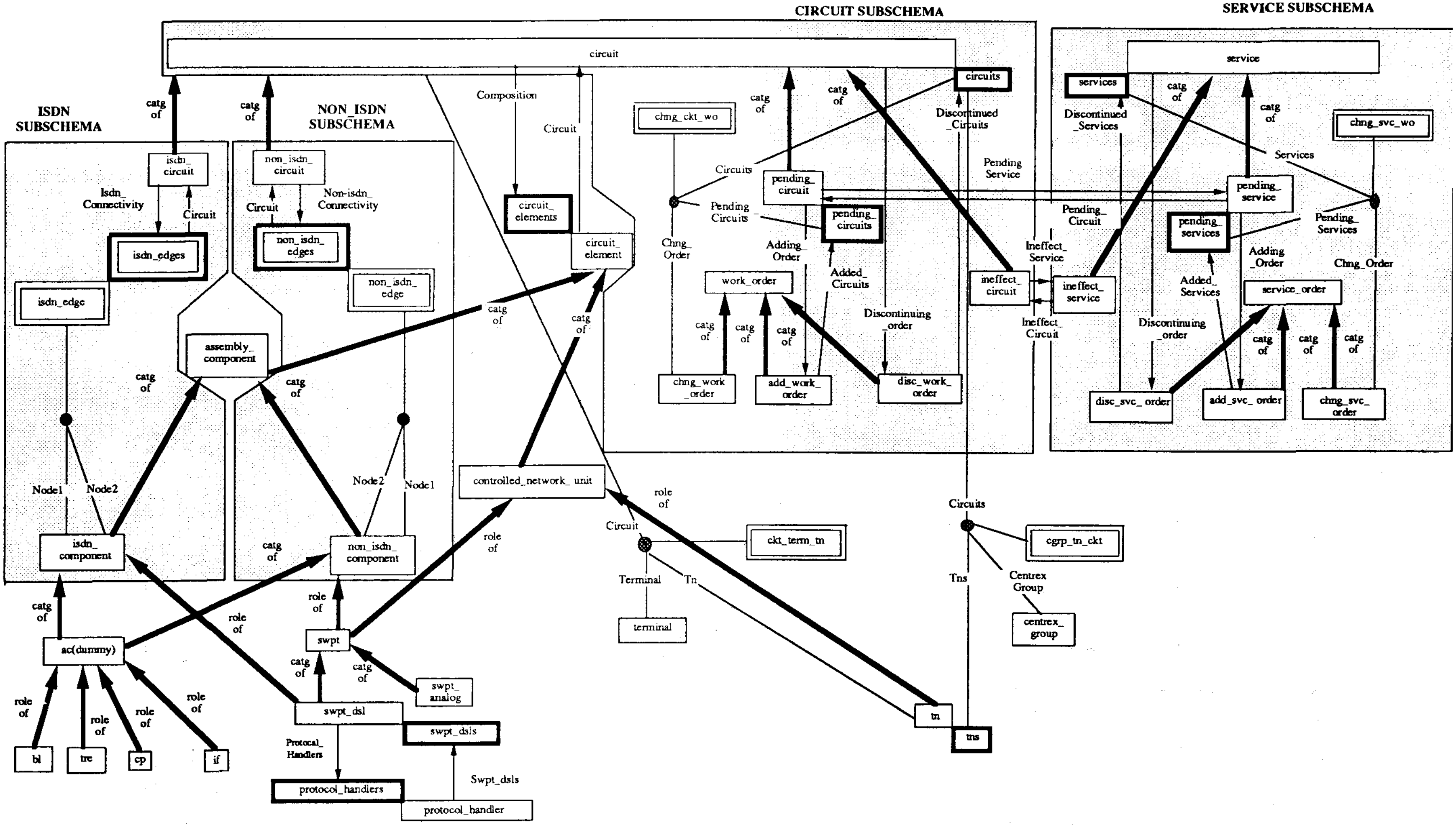

Figure 3. Semantic description of the Dual Model schema.

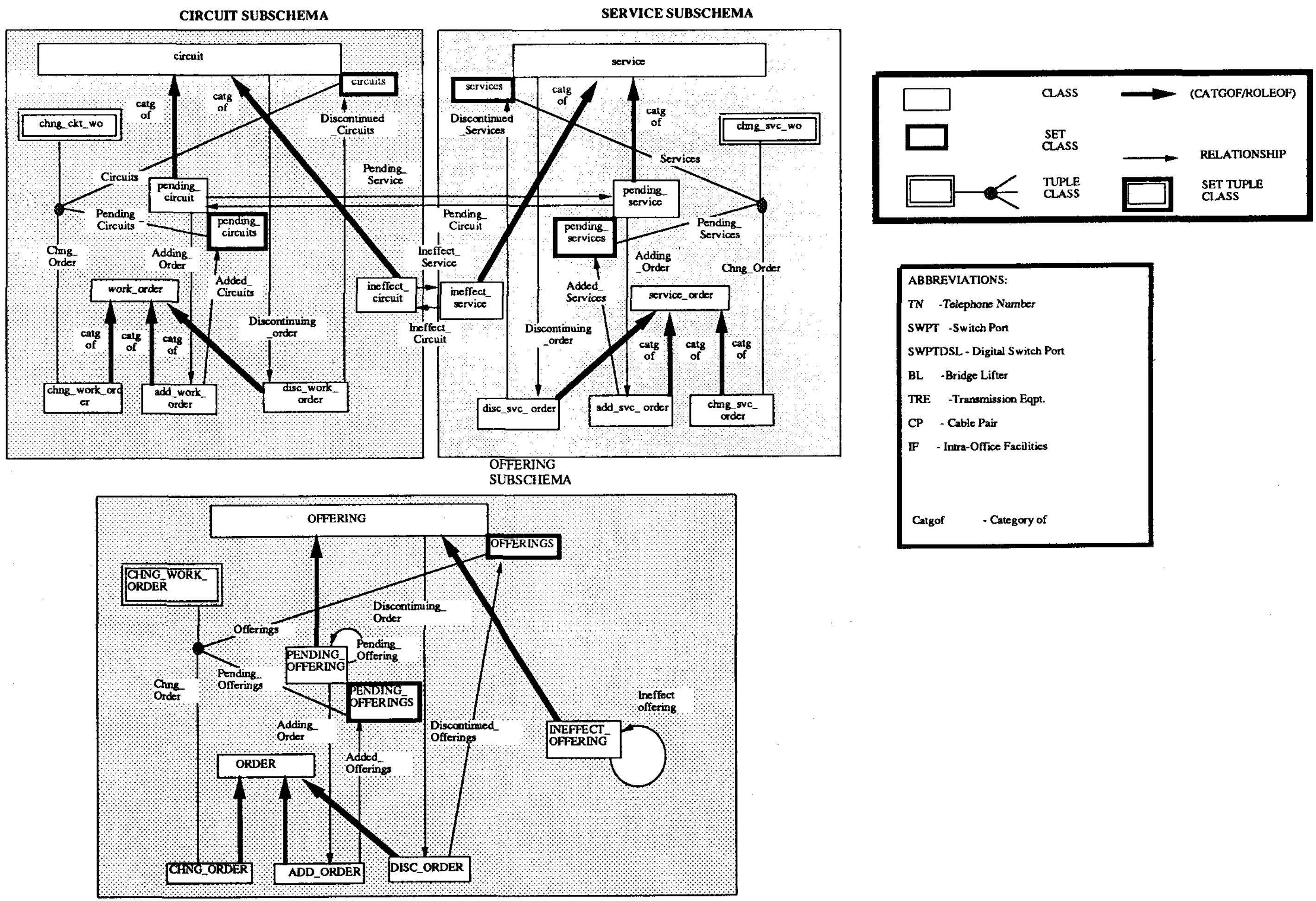

Figure 4. Semantic *CIRCUIT* and *SERVICE* subschemas and the structural *OFFERING* subschema.

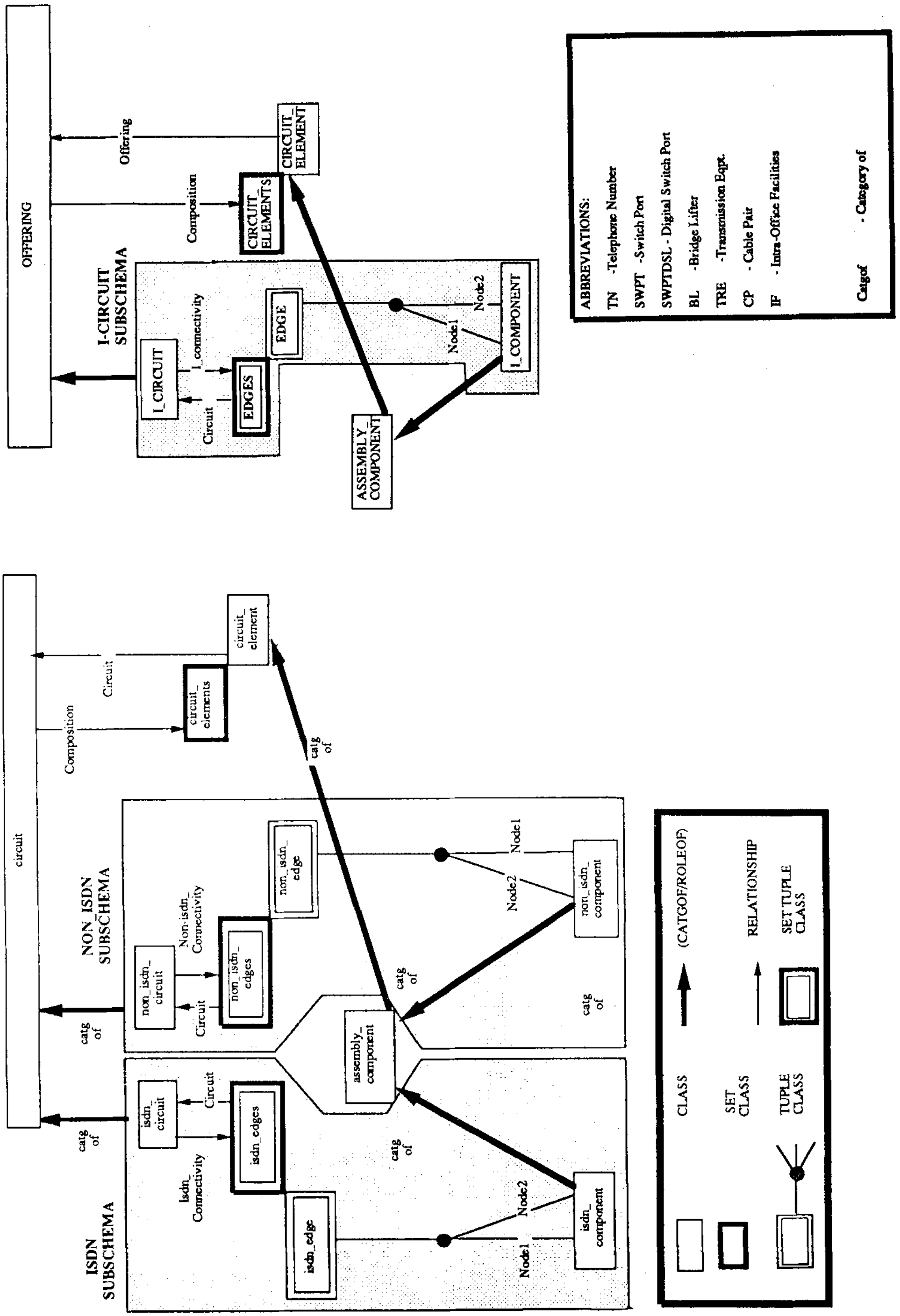

Figure 5. Semantic *ISDN* and *NON_ISDN* subschemas and the structural *I-CIRCUIT* subschema.

3.3. Generic Relations

Some generic relations, such as *subtypeof*, *setof*, and *memberof*, are structural and are defined between object types. A *subtypeof* generic relation connects a refined object type to a more general object type, enabling inheritance of the properties of the general type by the refined type. In the lower part of Figure 4, ADD_ORDER and CHNG_ORDER are subtypes of ORDER. The *subtypeof* generic relation specifies that the set of properties of the supertype is a subset of the set of properties of the subtype.

Whenever we need a relationship to refer to a set of objects, we can define an object type to represent this set. The connection of the set object type to the base object type is expressed with the structural *setof* and *memberof* generic relations. The object type CIRCUIT_ELEMENTS represents a *setof* CIRCUIT_ELEMENT (Figure 5), which is in turn a *memberof* the former. The Dual Model also supports semantic *setof* and *memberof* generic relations between classes.

The Dual Model contains two kinds of specialization generic relations, both of which are semantic. The first is *categoryof*, which relates the specialized class to the more general class when both are in the same context. The second is the *roleof* generic relation, which relates the specialized class to the more general class when the two are in different contexts. In the example, swpt_dsl (digital switchport) is a *categoryof* swpt (switchport) and a *roleof* isdn_component (Figure 3, bottom left).

Since structural generic relations induce a structural hierarchy, and semantic generic relations induce a semantic hierarchy, we also have two kinds of inheritance mechanisms: structural inheritance and semantic inheritance [22]. Many researchers (e.g., [4, 15]) assume that object-oriented databases are convenient for integration and code reusability due to their generalization capabilities expressed by the subclass hierarchy. The Dual Model refines the capability of specialization by using two hierarchies. Furthermore, the Dual Model offers the unique new technique for integration described in this paper, because it permits the assignment of one object type to several classes that are semantically different but share the same structure.

4. The Dual Model Semantic Representation of the Telecommunications Database

In this section we will comment in more detail on Figure 3, which shows the semantic representation of the Dual Model schema of the telecommunications application database corresponding to the ER schema shown in Figure 2. The Dual Model approach for dealing with ER-like relationships will be discussed. Finally, this section shows examples of the Dual Model treatment of methods.

In the ER model, a relationship is defined between two or more entities. In our object-oriented model, a relationship is one of the properties of a class and is represented graphically as an arrow directed from it to another class. If both directions of an ER relationship are relevant, then two relationships are represented in the object-oriented model, and two arrows pointing in opposite directions are drawn.

In the case of a one-to-many relationship, e.g., between add_svc_order and pending_service, we use a set class to represent sets of pending services, and the relationship

Added_Services points from add_svc_order to pending_services (shown under the *SERVICE* subschema of Figure 3). The other direction of the relationship, Adding_Order, points from each pending_service to add_svc_order. The case of a many-to-many relationship is demonstrated between the digital switchport swpt_dsl and protocol_handler (shown in the *NON_ISDN* subschema in Figure 3). Each of these two classes has a relationship to the set class of the other class, that is, to protocol_handlers and to swpt_dsls, respectively.

The ER model allows relationships among more than two entities. For example, the representation in Figure 2 contains several ternary relationships. To model a ternary or, in general, n-ary relationship, we create a tuple class which is composed of a sequence of several classes. Several tuple classes appear in Figure 3 to represent the ternary relationships of the ER schema. For instance, in Figure 3, in the *SERVICE* subschema, the tuple class chng_svc_wo represents the ternary relationship between services, pending_services and chng_svc_order. As expected, the structurally similar subschemas of the ER schema in Figure 2 have corresponding similar subschemas in the object-oriented representation in Figure 3. Structural integration allows us to exploit this similarity.

Figure 3 also demonstrates the use of path methods. Consider, for example, that a technician of a telephone company wants to install a circuit for a customer. He needs to know when this circuit should go into effect; however, this information is stored as attribute Date of the class add_svc_order that corresponds to this circuit. This information could be retrieved by adding a path method Service_Order_Date to the class pending_circuit. This path method would use a chain of relationships, terminated by an attribute, which graphically looks like:

[pending_circuit] $\xrightarrow{Pending\_Service}$ [pending_service] $\xrightarrow{Adding\_Order}$

[add_svc_order] $\xrightarrow{Date}$ DATETYPE.

The final attribute Date is not shown in Figure 3. This path method is a "frozen" and reusable record of a navigation through the schema. It is expressed entirely at the schema level.

If a customer calls to complain that a service does not work, then the customer-service representative might want to check the date of the work order for the corresponding circuit. To retrieve this information, it is necessary to write another method, which we will call Circuit_Order_Date. A graphical representation of this method would look like:

[pending_service] $\xrightarrow{Pending\_Circuit}$ [pending_circuit] $\xrightarrow{Adding\_Order}$

[add_work_order] $\xrightarrow{Date}$ DATETYPE.

Later on, we will use these two methods to show how structural integration can be applied to methods.

5. Structural Integration

Consider two sets of classes $C_1 = \{a_1, a_2, \ldots, a_n\}$ and $C_2 = \{b_1, b_2, \ldots, b_n\}$ of equal cardinality. For example, the *SERVICE* and the *CIRCUIT* subschemas in Figure 3 are such a pair. The essence of structural integration of two classes is to construct an object type that can be mapped into both classes. Structural integration of two sets of classes C_1 and C_2 requires therefore that pairs of classes (a, b) such that $a \in C_1$ and $b \in C_2$ can be found, for which it is possible to construct a common object type. If it is possible to construct such an object type for two classes a, b, then we say that they stand in *structural correspondence.*

There are two cases of structural correspondence, *full structural correspondence* and *partial structural correspondence*. Full structural correspondence is described in Section 5.1. Algorithms for checking full structural correspondence of two classes and two databases are presented in Section 5.3. The special case of attribute partial correspondence is presented in Section 5.4. Additional algorithms and further details of partial structural correspondence can be found in [8, 11].

5.1. Integration of Classes with Full Correspondence

For two corresponding classes a and b to have the same object type, there must exist a full structural correspondence between these two classes, i.e., between their sets of properties. Full correspondence for a pair of attributes means that their data types are identical. Full correspondence for relationships means that the referent classes have to be of the same object type. For both attributes and relationships, the selectors may be different. Fully corresponding generic relations must be of identical kind and point to classes of the same object type. The only exception to this is that we allow correspondence between *roleof* and *categoryof*, which we define to correspond.

Note that no explicit object type specification exists when the process of structural integration is attempted. Conditions must hold only between pairs of object classes.

Attributes, relationships, and generic relations can be written as ordered pairs, consisting of a selector and a data type (or object type, or class). For instance, in Figure 3 (right side) the relationship Discontinuing_order points to the class disc_svc_order. It can be viewed as the pair (Discontinuing_order, disc_svc_order). We will use the LISP convention to extract the first element from a pair with the operation CAR, and the second element with the operation CDR [26]. Then we can define a number of useful operators as follows.

$$\text{SELECTOR} \equiv \text{CAR}$$

For simplicity, we will omit the inner pair of parentheses. For example,

SELECTOR((Discontinuing_order, disc_svc_order)) =
SELECTOR(Discontinuing_order, disc_svc_order) = Discontinuing_order.

The function DATATYPE expects as its argument a pair that describes an attribute. It is undefined for any other kind of argument.

$$\text{DATATYPE} \equiv \text{CDR}$$

The function CLASS expects as its argument a pair that describes a relationship or generic relation defined for a class. It is undefined for any other kind of argument.

$$\text{CLASS} \equiv \text{CDR}$$

The function OBJECTTYPE expects as its argument a pair that describes a relationship or generic relation defined for an object type. Again, it is undefined for any other kind of argument.

$$\text{OBJECTTYPE} \equiv \text{CDR}$$

The function RELATIONNAME expects as its argument a pair that describes a generic relation. It is undefined for any other kind of argument.

$$\text{RELATIONNAME} \equiv \text{CAR}$$

Formally, let the class a (b) have a set $\{x_i\}$ ($\{y_i\}$) of attributes, a set $\{r_j\}$ ($\{s_j\}$) of relationships, a set $\{m_k\}$ ($\{n_k\}$) of methods, and a set $\{g_l\}$ ($\{h_l\}$) of generic relations to other classes. The classes a and b have *full structural correspondence* if:

1. There exists a one-to-one correspondence between the sets of attributes $\{x_i\}$ and $\{y_i\}$ such that x_i corresponds to y_i if DATATYPE(x_i) = DATATYPE(y_i).
2. There exists a one-to-one correspondence between the sets of relationships $\{r_j\}$ and $\{s_j\}$ such that it must be possible to construct a common object type for CLASS(r_j) and CLASS(s_j).
3. There exists a one-to-one correspondence between the sets of methods $\{m_k\}$ and $\{n_k\}$ such that when m_k is a path method that defines a path going through the sequence a_1, a_2, . . . , a_s of classes, and n_k defines a similar path through b_1, b_2, . . . , b_t, then the following conditions hold: (1) $s = t$; and (2) it is possible to construct a common object type for a_i and b_i, $1 \leq i \leq s$.
4. There is one-to-one correspondence between the sets of generic relations $\{g_l\}$ and $\{h_l\}$ such that either RELATIONNAME(g_l) = RELATIONNAME(h_l) or both relation names are members of the set $\{$*roleof*, *categoryof*$\}$. In both cases, it must be possible to construct a common object type for CLASS(g_l) and CLASS(h_l).

In Figure 3, subschema *SERVICE* describes service orders and services. Subschema *CIRCUIT* describes work orders and circuits. Pairs of classes with structural correspondence and the names of their corresponding object types are listed in Table 1. The name of the object type was derived by extracting the common parts of the names of the two corresponding classes, the exception being *service* and *circuit* whose object type is OFFERING.

Table 1.

SERVICE Subschema Class	*CIRCUIT* Subschema Class	Object Type
service_order	work_order	ORDER
add_svc_order	add_work_order	ADD_ORDER
disc_svc_order	disc_work_order	DISC_ORDER
chng_svc_order	chng_work_order	CHNG_ORDER
service	circuit	OFFERING
services	circuits	OFFERINGS
ineffect_service	ineffect_circuit	INEFFECT_OFFERING
pending_service	pending_circuit	PENDING_OFFERING
pending_services	pending_circuits	PENDING_OFFERINGS

Figure 4 shows the structural integration of the *SERVICE* and *CIRCUIT* semantic subschemas of Figure 3 by presenting the *OFFERING* structural subschema. For each pair of corresponding classes in these two subschemas, the *OFFERING* subschema contains a common object type, as shown in Table 1. Each occurrence of the semantic *categoryof* relation in Figure 3 is replaced in Figure 4 by the *subtypeof* structural relation in the structural *OFFERING* subschema.

Using Table 1 and Figure 4 it is easy to see that there is an almost full correspondence of the relationships and generic relations between the corresponding classes of the two subschemas. In addition there are two relationships, Pending_Service and Pending_Circuit, between the classes in the different subschemas. The two relationships were used in the two previously introduced methods Service_Order_Date of the class `pending_circuit` and Circuit_Order_Date of the class `pending_service` which correspond to one another. The two methods can be integrated by a structural path method which we will call Offering_Order_Date. This method needs to be defined in the object type PENDING_OFFERING and will now be shown graphically.

[PENDING_OFFERING] $\xrightarrow{\textit{Pending\_Offering}}$ [PENDING_OFFERING] $\xrightarrow{\textit{Adding\_Order}}$

[ADD_ORDER] $\xrightarrow{\textit{Date}}$ DATETYPE.

This is an interesting path method, as it contains a connection from an object type to itself. We call such a path method *reflexive*. There are several incoming external connections that appear only for the *CIRCUIT* subschema, but they do not disturb the process of structural integration, as they are defined for classes that are not integrated. There is one outgoing external connection that appears only for the *CIRCUIT* subschema, namely Composition. This one relationship stands in the way of a full structural correspondence and cannot be handled with the techniques developed in this article. We will show the treatment of Composition, without discussing the formal techniques necessary for partial structural correspondence. Those can be found in [11].

5.2. The Mapping between Object Types and Classes

Object types and classes are not independent. The specification (i.e., the code) of a class contains the name of the object type that summarizes its structure, prefixed by the keyword *objecttype*. The details of the dependency are expressed by a mapping M from the set P of properties of the object type to the set Q of properties of the class. This one-to-one mapping from P onto Q [16] identifies for each $p \in P$ the corresponding $q \in Q$. If an object type (A) has only one class (a) associated, then the specification of the class a is sufficient and the object type A can be defined by an algorithm (see [22]).

A *categoryof* generic relation always has a corresponding *subtypeof* relation [22]. A *roleof* generic relation may or may not have a corresponding *subtypeof* relation. If a *roleof* has a corresponding *subtypeof*, then the *roleof* is annotated by *subtypeof* in the class description.

Only *roleof* generic relations with corresponding *subtypeof* are relevant for structural integration, because structural integration is realized by sharing of object types by classes and *subtypeof* is the only specialization relation defined for object types. Structural integration allows a correspondence of *roleof* and *categoryof*, as both are represented in the structural schema by a *subtypeof* relation. A *roleof* generic relation without an associated *subtypeof* is ignored in the process of structural integration. The two object types corresponding to the two classes connected by the *roleof* are not connected at all. In our example all *roleof* relations happen to lie outside the subschemas that are used for structural integration, and therefore these concerns are not applicable to it.

*5.2.1. **Conditions for Mapping from an Object Type to One Class.*** Following are the conditions for a mapping M that must be satisfied when an object type A has one object class a associated with it.

Selector: The corresponding properties of the object type and the class have the same selector. For example, an attribute in the structural description has a corresponding attribute with the same name in the semantic description. The same is true for all other properties.

Attributes: Corresponding attributes have the same data type.

Relationships: For every relationship r from an object type A to an object type B, there is a relationship r' of the class a that refers to a class b having an object type B.

Generic specialization relations: For every *subtype* generic relation g from an object type A to an object type B, the class a may have either a *categoryof* or a *roleof* relation to a class b having an object type B, but there might be no connection at all.

Generic set relations: For every *memberof* (*setof*) generic relation g from an object type A to an object type B, the corresponding generic relation g' is a *memberof* (*setof*) relation from the class a to a class b having the object type B.

Path methods: For methods we will limit ourselves to "path methods." The formal condition for a mapping is then as follows: For every path method m from an object type A_1

referring to a sequence of object types $A_2, A_3, \ldots, A_n$, the path method m' of the class a_1 refers to a sequence of classes $a_2, a_3, \ldots, a_n$ such that the class a_i, $1 \leq i \leq n$, has the object type A_i, $1 \leq i \leq n$. Furthermore, let p_i be the relationship or generic relation of A_i such that $p_i(A_i) = A_{i+1}$, $1 \leq i \leq n$, and let q_i be the relationship or generic relation such that $q_i(a_i) = a_{i+1}$, $1 \leq i \leq n$; then the mapping from A_i to a_i must satisfy a mapping $M'(p_i) = q_i$.

(Note: This mapping M′ is identical to M only for the first step of the path, because this first step is a relationship or generic relation defined in the class for which we define M. For all other relationships or generic relations of the path, M′ depends on the classes in which the relationships are defined.)

As an example for a **Relationship**, consider Figures 3 and 6. The class `swpt_dsl` has a relationship Protocol_Handlers to the class `protocol_handlers` (Figure 3, bottom left). In Figure 6 (bottom left) there is a relationship of the same name (Protocol_Handlers) referring to the object type PROTOCOL_HANDLERS. PROTOCOL_HANDLERS is the object type that is associated with the class `protocol_handlers`.

An example for a **Generic Set Relation** can be seen in the same figures. The *setof* relation from `swpt_dsls` to `swpt_dsl` (Figure 3) corresponds to the *setof* relation to the object type SWPT_DSL, which is the object type associated with the object class `swpt_dsl`. The *setof* relation is shown by the shared corners of the two classes.

5.2.2. Conditions for Mappings from an Object Type to Several Classes. Consider now the case where two classes a_1 and a_2 have the same object type A.[4] For this case we have to relax some of the previous conditions. Let P be the set of properties of A. Let Q_1 and Q_2 be the sets of properties of a_1 and a_2, respectively. In such a case we have the mappings $M_1: P \rightarrow Q_1$ and $M_2: P \rightarrow Q_2$ satisfying the following conditions. Let $M_1(p) = q_1$ and $M_2(p) = q_2$, where $p \in P$, $q_1 \in Q_1$ and $q_2 \in Q_2$.

Property kind: The properties p, q_1, and q_2 should be of the same kind, i.e., they all should be attributes, or they all should be relationships, or they all should be methods, or they all should be generic relations.

Selectors: The selectors of the properties p, q_1, and q_2 are not necessarily identical.

Attributes: Let p, q_1, and q_2 be attributes in pair notation. Then $\text{DATATYPE}(p) = \text{DATATYPE}(q_1) = \text{DATATYPE}(q_2)$.

Relationships: Let p be a relationship from A to an object type B, q_1 be a relationship from a_1 to a class b_1, and q_2 be a relationship from a_2 to a class b_2. Then b_1 and b_2 should have the same object type B.

Generic specialization relations: Let p be a *subtypeof* relation from A to an object type B. Then each of the relations q_1 and q_2 is either a *categoryof* or a *roleof* relation to the classes b_1 and b_2, respectively, such that b_1 and b_2 have the object type B; alternatively q_1 and q_2 might be undefined (nonexistent).

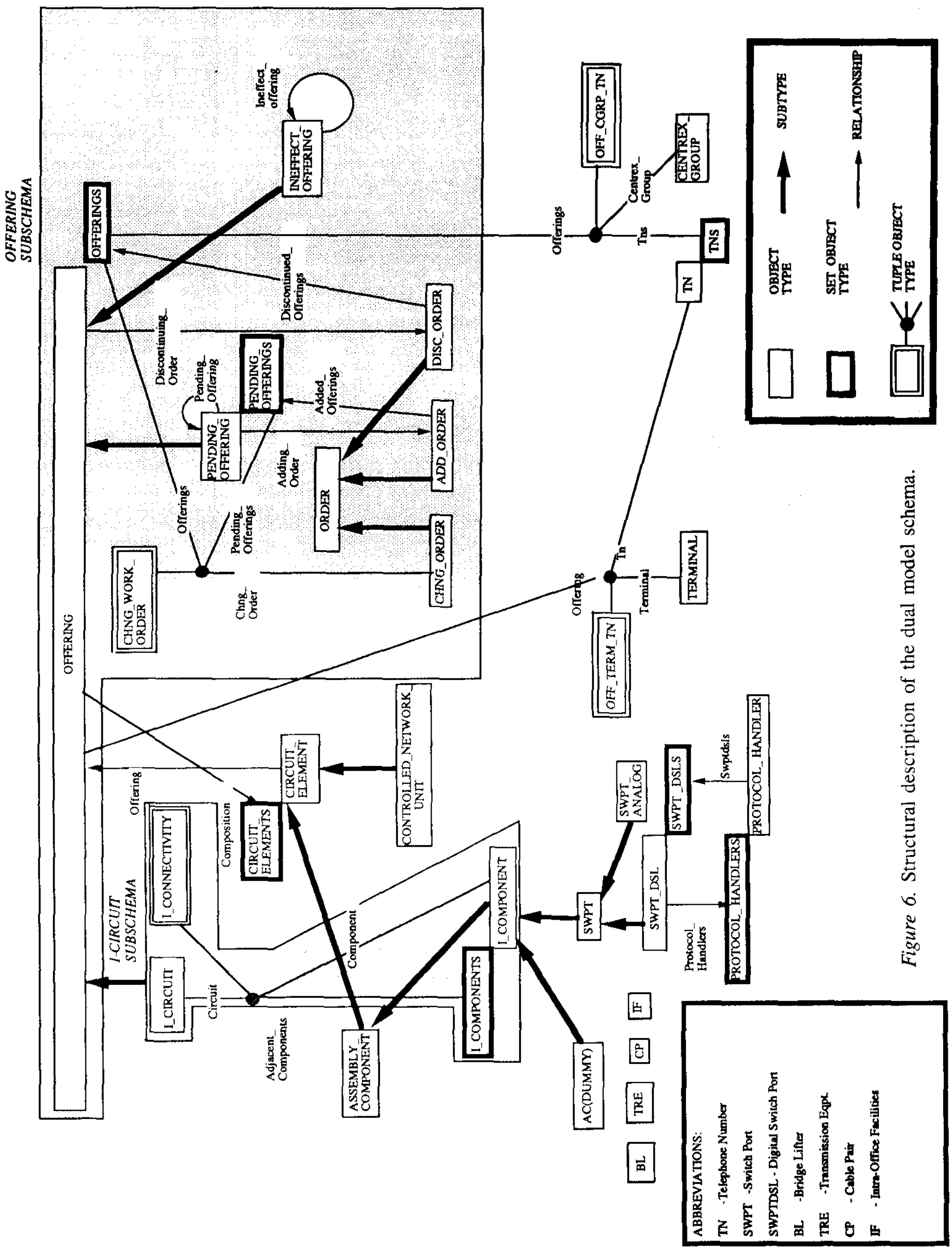

Figure 6. Structural description of the dual model schema.

Generic set relations: Let p be a *memberof* or a *setof* relation from A to an object type B. Then the relations q_1 and q_2 are *memberof* or *setof* relations to b_1 and b_2, respectively, such that b_1 and b_2 have the object type B.

Path methods: Let p be any relationship or generic relation at position k in a path method from A_1 to an object type A_n, so that p connects two object types A_k and A_{k+1}. Then q_1 connects two classes a_k and a_{k+1}, and q_2 connects two classes b_k and b_{k+1} such that a_k and b_k have the object type A_k and a_{k+1} and b_{k+1} have the object type A_{k+1}, and both q_1 and q_2 are at position k in their respective path methods.

A **Generic set relation** is demonstrated in Figure 4. A *setof* relation points to the object type PENDING_OFFERING in the structural *OFFERING* subschema. Therefore, there exist *setof* relations from pending_circuits to pending_circuit in the *CIRCUIT* subschema and from pending_services to pending_service in the *SERVICE* subschema. The classes pending_circuit and pending_service have the common object type PENDING_OFFERING.

The interface between object types in the structural subschema *OFFERING* and the classes in the subschemas *SERVICE* and *CIRCUIT* is given by the respective mappings. For example, the object type PENDING_OFFERING has two corresponding classes, pending_service and pending_circuit. Let P, Q_1 and Q_2 be the sets of properties of PENDING_OFFERING, pending_service and pending_circuit, respectively. We show parts of the mappings M_1: $P \rightarrow Q_1$ and M_2: $P \rightarrow Q_2$ in Tables 2 and 3, for relationships and relations, respectively.

The graphic representation is a powerful presentation and learning tool. However, for practical database use, code is specified by assigning to every class in the object-oriented database its relationships, attributes, methods, and generic relations. In the Dual Model, this description is split into a structural description of the object type and a semantic description of the class. These two parts are shown in Table 4. The semantic information is written in the right column, and the object type information is in the left column. The attributes in the object type PENDING_OFFERING are only hinted at by "$attr_1$," "$attr_2$," etc.

The mappings from the object type PENDING_OFFERING to the two object classes pending_service and pending_circuit were given in Table 2. In this example, the

Table 2.

M_1: (Adding_Order, ADD_ORDER) → (Adding_Order, add_svc_order)
M_2: (Adding-Order, ADD_ORDER) → (Adding_Order, add_work_order)
M_1: (Pending_Offering, PENDING_OFFERING) → (Pending_Circuit, pending_circuit)
M_2: (Pending_Offering, PENDING_OFFERING) → (Pending_Service, pending_service)

Table 3.

M_1: (subtype, OFFERING) → (categoryof, service)
M_2: (subtype, OFFERING) → (categoryof, circuit)
M_1: (memberof, PENDING_OFFERINGS) → (memberof, pending_services)
M_2: (memberof, PENDING_OFFERINGS) → (memberof, pending_circuits)

Table 4.

Object Type Description	Semantic Class Description
objecttype PENDING_OFFERING **subtypeof:** OFFERING **memberof:** PENDING_OFFERINGS **attributes** $attr_1$ $attr_2$ . . . . $attr_n$ **relationships:** Adding_Order: ADD_ORDER Pending_Offering: PENDING_OFFERING	**class** pending_service **objecttype:** PENDING_OFFERING **categoryof:** service **memberof:** pending_services **relationships:** Adding_Order: add_svc_order Pending_Circuit: pending_circuit **class** pending_circuit **objecttype:** PENDING_OFFERING **categoryof:** circuit **memberof:** pending_circuits **relationships:** Adding_Order: add_work_order Pending_Service: pending_service

attributes are specified only once in the object type, but are known to both classes. This is comparable to generalization, where attributes are specified with a single class and known to all of its subclasses.

Structural integration of the *ISDN* and *NON-ISDN* subschemas is similar to that of the *CIRCUIT* and *SERVICE* subschemas, and is shown in Table 5 and Figure 5.

The *ISDN* subschema models the isdn circuit as a graph with components as nodes and wires as edges connecting nodes. An isdn_circuit is a *categoryof* circuit since it is a special kind of circuit. It has a relationship Isdn_connectivity to the set isdn_edges of the circuit. This set in turn has the reverse relationship Circuit to the isdn_circuit and a *setof* generic relation to the tuple class isdn_edge. The latter is a tuple class since it is composed of two isdn_components as the two end nodes of the edge. This is an interesting case of a tuple class since the same class isdn_components appears twice in it. Finally, an isdn_component is *categoryof* an assembly_component. The structure of the NON_ISDN subschema is a mirror image of the *ISDN* subschema. Their structural integration is expressed in the structural *I-CIRCUIT* subschema (Figure 5).

Figure 6 shows a complete diagram for the structural representation of the Dual Model for the telecommunications database from Figure 3. For each class in Figure 3, there is

Table 5.

ISDN Subschema Class	*NON-ISDN* Subschema Class	Object Type
isdn_circuit	non_isdn_circuit	I_CIRCUIT
isdn_component	non_isdn_component	I_COMPONENT
isdn_edge	non_isdn_edge	EDGE
isdn_edges	non_isdn_edges	EDGES

an object type in Figure 6 with the same name, except for the pairs of structurally similar classes (i.e., classes with correspondence) where one object type replaces both similar classes (as indicated in Tables 1 and 5). The generic relations in Figure 6 are structural and not semantic, thus we do not show *roleof* or *categoryof*, but we show *subtypof* between object types. As noted before, if class a is a *categoryof* class b, then the corresponding object type A must be a *subtypeof* the corresponding object type B. For the *roleof* generic relation, this may or may not be the case. It is the case if we want the specialized type to inherit all the properties of the general type. This does not occur in our example. Examples for both possibilities are given in [22]. Note that the structural representation does not replace the semantic representation; rather it shows the structural aspects of the database. The approach used in the Dual Model is to give the user both representations. The integration is represented by the structural schema of object types and two mappings to the schemas of classes.

5.3. Algorithms for Checking Full Structural Correspondence of Classes and Databases

We start this section with an algorithm to check whether two given classes satisfy full structural correspondence. This algorithm implements a test whether the conditions of Section 5.1 are met.

PROCEDURE CORRESPONDENCE(a, b: class)

1. IF the number of attributes in a and b is not equal
   THEN exit(a, b)
      /* All exits in this algorith are failure exits. */
   IF the number of attributes of any given data type in a and b is not equal
   THEN exit(a, b)
2. Consider the *categoryof* and *roleof* generic relations of a and b.

2.a IF their numbers are not equal THEN exit(a, b).

2.b IF a and b both have one such relation to classes a_1 and b_1 respectively,
   AND a_1 and b_1 do not have identical object types
   THEN exit(a, b)
      /* This is the case of Single Inheritance. */

2.c ELSE

/* This the case of Multiple Inheritance. a has many specialization relations which may be either *categoryof* or *roleof* relations to $a_1, a_2, \ldots, a_m$, and b has many specialization relations which also can be either *categoryof* or *roleof* relations to $b_1, b_2, \ldots, b_m$. In this step we look for a one-to-one matching between $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots, b_m$. Note that in structural integration a *categoryof* may match a *roleof*! */

```
FOR k := 1 TO m DO
   match[k] := 0        /* The array match is used to record the correspondence. */
FOR i := 1 TO m DO
   {flag := FALSE       /* flag is used to indicate whether a_i is matched. */
   k := 1
```

```
WHILE k ≤ m AND NOT flag DO
  {IF match[k] = 0 THEN        /* b_k is free for matching. */
    IF a_i and b_k have a common object type THEN
      {match[k] := i        /* So, b_k cannot match a second a_j. */
      flag := TRUE}
    ELSE k := k + 1}
IF NOT flag THEN exit(a, b)}
  /* No full correspondence possible since a_i was not matched. */

OUTPUT match[1..m]. /* This is the success case of full correspondence. */
  /* If the array contains at u the number v that means that b_u corresponds to a_v. */
```

3. Consider the *setof* relations of a and b (there exists, at most, one).
   The treatment is the same as in step 2 (a and b only).
4. Consider the *memberof* relations of a and b (there may be more than one).
   The treatment is the same as in step 2.
5. Consider the relationships of a and b.
   The treatment is the same as in step 2.

The complexity of the CORRESPONDENCE(a, b) algorithm is $O(c^2)$ where

c = maximum(#relationships, #categoryof + #roleof relations, #memberof relations).

The # operator returns the number of connections of the given kind, i.e., #roleof returns the number of *roleof* connnections of the class a. If CORRESPONDENCE(a, b) completes successfully, then we can apply the algorithm STRUCTURAL_INTEGRATION(a, b, A) (presented in [8, 11]), which creates a common object type A for the fully corresponding classes a and b, using the output of the "match" array for the correpsondence between the relations and relationships of the two classes.

We can use the algorithm CORRESPONDENCE(a, b) to find pairs of corresponding classes from both given databases. However, the order of processing the classes may have an impact. Suppose for example, that class a_1 (b_1) has a *categoryof* relation to class a_2 (b_2). Suppose further that none of these classes has more connections to other classes and that the attributes of a_1 and b_1 (a_2 and b_2) have full structural correspondence. Then if we apply CORRESPONDENCE(a_1, b_1), the matching will fail due to the *categoryof* relation since a_2 and b_2 do not yet have a common object type [see step 2 in procedure CORRESPONDENCE(a, b)]. However, if we start with CORRESPONDENCE(a_2, b_2) followed by CORRESPONDENCE(a_1, b_1), then both applications of CORRESPONDENCE will be successful since while applying CORRESPONDENCE(a_1, b_1) the classes a_2 and b_2 are already of the same object type A_2. Thus, a_1 and b_1 have the same object type A_1. As a matter of fact, A_1 will have a subtype relation to A_2.

If a_1 (a_2) and b_1 (b_2) have cyclic connections, i.e., there is a directed path from a_1 (a_2) to b_1 (b_2) and vice versa, then no order of processing will help. That is, a_1 and a_2 may potentially have the same object type A_1, and b_1 and b_2 may potentially have the same object type A_2, but due to the cyclic nature of the connections of the classes it is impossible to recognize this fact with the CORRESPONDENCE procedure applied in any order. For an algorithm for structural integration of cyclic schemas see [8, 11].

In order to process the classes of each database **Da** which do not participate in cyclic subschemas and gain the possible results from applying the CORRESPONDENCE algorithm, we need to reorder the classes in each database as follows.

PROCEDURE REORDER (Da)
The REORDER procedure applies topological sort
to the acyclic portion of the database.

Topological sort [1] is a well-known technique which does not need to be repeated here. Now we can present an algorithm DB_INTEGRATE for finding and creating common object types for classes with full correspondence of two databases **Da** and **Db**. Let **Da** have m classes $a_1, a_2, \ldots, a_m$ and **Db** have n classes $b_1, b_2, \ldots, b_n$. Let **DA** be the set of the explicitly defined object types A_k for the integrated database.

PROCEDURE DB_INTEGRATE (Da, Db, DA)

```
REORDER(Da)        /* Call to the previous Procedure */
REORDER(Db)
k := 0
FOR i := 1 TO m DO
   FOR j := 1 TO n DO
      {CORRESPONDENCE(a_i, b_j)
      IF CORRESPONDENCE(a_i, b_j) returns successfully
      THEN {k := k + 1; STRUCTURAL_INTEGRATION(a_i, b_j, A_k)}}
```

For the classes which were not matched, their object types are still defined implicitly in the integrated database as they were in **Da** and **Db** prior to the structural integration. The complexity of this algorithm is $O(mnc^2)$ where c is defined as before.

5.4. Structural Integration of Classes with Attribute Partial Correspondence

Full structural correspondence is a good basis for developing a general theory of correspondence and also a good vehicle for introducing the concept to the reader. However, it is not a practical case for real databases. Surprisingly enough, it was possible to solve most of the problems of the telecommunications database schema from Figure 2 by a formalism which is intermediate in complexity between full structural correspondence and the completely general case. This case is referred to as *attribute partial correspondence* and will be formalized in this section.

Attribute partial correspondence implies that there is a full correspondence in the relationships and generic relations. Because relationships and generic relations always involve two schema elements, they appear more important for structural integration than attributes which affect only one schema element. Therefore one would probably attempt structural integration of two classes standing in attribute partial correspondence, even if there are quite a few differences between the sets of attributes. However, this decision is up to the designer of the database and requires some understanding of the application.

To develop the notion of attribute partial correspondence, we first need to define *partial structural correspondence* between two classes a and b. Let Q_a and Q_b be the sets of properties of a and b. Let $\mathcal{Q}_a \subset Q_a$ and $\mathcal{Q}_b \subset Q_b$ be the sets of properties for which there is a one-to-one correspondence, as defined for full structural correspondence. Then the classes a and b are defined to stand in partial structural correspondence if $\mathcal{Q}_a \neq \emptyset$ and $\mathcal{Q}_b \neq \emptyset$. Another way of looking at this definition is to assume a superclass a_0 of a with the set $\mathcal{Q}_a$ of properties and a superclass b_0 of b with the set $\mathcal{Q}_b$ of properties, such that the classes a_0 and b_0 have a full structural correspondence.

One should not attempt to integrate a and b, unless $|\mathcal{Q}_a| \approx |Q_a|$ and $|\mathcal{Q}_a| \approx |Q_b|$. If the sets of properties $Q_a - \mathcal{Q}_a$ and $Q_b - \mathcal{Q}_b$ include attributes only, we talk about *attribute partial correspondence.* If they include relationships only, we talk about *relationship partial correspondence.* To summarize, attribute partial correspondence means that some attributes do not correspond but everything else does. We concentrate here on attribute partial correspondence, which describes our application (almost) perfectly.

In the telecommunications database examples of Figure 3 there exists attribute partial correspondence for each pair of corresponding classes in similar subschemas except for `service` and `circuit`, which differ in that `circuit` has one extra relationship called Composition. The extra generic relations or tuple-type relations which refer to `circuit` do not matter for this purpose because they point to `circuit` and not away from it. For each other pair of classes there exists full correspondence of relationships and generic relations. On the other hand, there are differences in the sets of attributes for most pairs of classes in the subschemas.

The difference in the Composition relationship can be solved by techniques defined in [11] which extend those introduced in this article. Returning to attributes, consider the corresponding classes `circuit_order` and `service_order` in the diagram of Figure 3. Each one has 30 attributes, and 18 of them are common to both classes. Now how do we define the attributes for the object type ORDER, which is the object type of both classes? In general, suppose we are given two classes, a and b, which exhibit attribute partial correspondence. Let X_a be the set of *m attributes of a*, and let X_b be the set of n attributes of b. Let $C \subset X_a$ and $D \subset X_b$ be the subsets of the attributes for which there exists a one-to-one correspondence. (We are not using "$\subseteq$" because we are dealing with partial correspondence. As a matter of fact, any one of the two "$\subset$" operators may be a "$\subseteq$," but not both at the same time.) The definition of the properties of an object type A that integrates two classes a and b requires only the specification of X_A, the attributes of A, since a and b stand in attribute partial correspondence. In other words, relationships, generic relations, and methods already exhibit full structural correspondence. The set of attributes is defined as follows:

$$X_A = X_{common} \cup Y \cup Z,$$

where the set X_{common} contains an attribute for each pair (x, y) of corresponding attributes $x \in C$, $y \in D$. Y and Z are defined as

$$Y = X_a - C \text{ and } Z = X_b - D.$$

If $x \in C$ and $y \in D$ are two corresponding attributes, then it must be the case that $\text{DATATYPE}(x) = \text{DATATYPE}(y)$. The two attributes may or may not agree in their selectors. If they do agree, we will use the common selector for the attribute of the object type. Formally, if $\text{SELECTOR}(x) = \text{SELECTOR}(y)$, then we define an attribute $w \in X_{common}$ such that

$$w = (\text{SELECTOR}(x), \text{DATATYPE}(x)).$$

If they disagree, i.e., if $\text{SELECTOR}(x) \neq \text{SELECTOR}(y)$, then we are free to define an attribute $w \in X_{common}$.

$$w = (\text{SELECTOR}(w), \text{DATATYPE}(x))$$

with no constraint, i.e.,

$$\text{SELECTOR}(w) = \text{SELECTOR}(x) \text{ or}$$

$$\text{SELECTOR}(w) = \text{SELECTOR}(y) \text{ or}$$

$$\text{SELECTOR}(w) \neq \text{SELECTOR}(x) \text{ and } \text{SELECTOR}(w) \neq \text{SELECTOR}(y)$$

This process has to be performed for each such pair of attributes (x, y) satisfying the above conditions. For each attribute $w \in (X_a - C)$, we define for Y an attribute $y = (\text{SELECTOR}(y), \text{DATATYPE}(y))$. For each attribute $w \in (X_b - D)$, we define for Z an attribute $z = (\text{SELECTOR}(z), \text{DATATYPE}(z))$.

Now we specify a mapping M_1 from the set X_A of attributes of A to the set X_a of attributes of a, and a mapping M_2 from X_A to X_b. The cardinality of X_A is

$$|X_A| = m + n - |C|.$$

Thus, M_1 and M_2 are both mappings from a larger set to a smaller set and are defined by specifying how they map the individual subsets of X_A (i.e., X_{common}, Y and Z).

For each attribute $w \in X_{common}$, let (x, y), $x \in X_a$, $y \in X_b$ be the pair of corresponding attributes used in defining the attribute w. Then

$$M_1\text{: } (\text{SELECTOR}(w), \text{DATATYPE}(w)) = (\text{SELECTOR}(x), \text{DATATYPE}(x))$$

$$M_2\text{: } (\text{SELECTOR}(w), \text{DATATYPE}(w)) = (\text{SELECTOR}(y), \text{DATATYPE}(y)).$$

For each attribute $y \in Y$, the mappings are

$$M_1\text{: } (\text{SELECTOR}(y), \text{DATATYPE}(y)) = (\text{SELECTOR}(y), \text{DATATYPE}(y))$$

$$M_2\text{: } (\text{SELECTOR}(y), \text{DATATYPE}(y)) = \text{NULL}.$$

For each attribute $z \in Z$, the mappings are

$$M_1(\text{SELECTOR}(z),\ \text{DATATYPE}(z)) = \text{NULL}$$

$$M_2(\text{SELECTOR}(z),\ \text{DATATYPE}(z)) = (\text{SELECTOR}(z),\ \text{DATATYPE}(z)).$$

In this way we obtain two mappings, M_1, which is one-to-one from X_A onto X_a and M_2, which is one-to-one from X_A onto X_b. To summarize, our approach enables a mapping to both sets of attributes of the classes a and b by inserting the corresponding attributes of both a and b into the object type A. This technique should be used carefully since it can be abused by integrating two classes that have little in common.

6. Conclusions

We discussed a technique that allows sharing of database structures (schema objects) even when they have different semantics. This leads to a better understanding of data and to sharing of data and methods. Throughout the article, we used the example of an existing large telecommunications application database to investigate the complexity and scalability of our technique.

Structural integration would not be possible without the Dual Model, which permits a separation of the structure and semantics of an object-oriented database. A specification in the Dual Model consists of two schemas. The building blocks of the semantic specification are called classes, the building blocks of the structural specification are called object types. Every class must have exactly one corresponding object type, but one object type may have several corresponding classes. It is exactly that last characteristic of the Dual Model that is the basis for the theory of structural integration.

The Dual Model can integrate two classes even if it is impossible to find a common superclass for them, because a single object type can correspond to several semantically different classes, as long as they are structurally similar. As such, structural integration makes use of a novel form of semantic relativism which is provided by the Dual Model, and permits integration in cases where other known methods fail.

The simplest case of structural integration occurs when the two sets of properties of the two classes to be integrated correspond perfectly to each other in their numbers and kinds. This is referred to as full structural correspondence. However, even for full structural correspondence, the properties may differ in their selectors and order, and the classes to which relationships point may be different as long as these classes are associated with the same object type. Two classes are then integrated by defining a common object type and two mappings from the properties of this object type to the properties of the classes. The more complicated case of integration with partial correspondence has been defined. The special case of attribute partial correspondence, where there are only differences in attributes, has been discussed.

Besides extending the range of cases where integration is possible, structural integration has the following additional advantages. The integration process allows sharing of attributes, relationships, and methods, thus contributing to compact representations and software reusability. Especially for methods it is possible to achieve a considerable amount of savings in specification.

Acknowledgements

We thank Ashok Ingle, Gene Wuu, Howard Marcus, Pat Quigley, Bob Davis, Veena Teli, and Nevil Patel for helpful discussions, and for helping us understand the telecommunications application. The schema in Figure 2 is loosely based on concepts used in the SWITCH™ application modeled by Ashok Ingle. Mike Halper has proofread this article. Thoughtful referee comments are gratefully acknolwedged.

Notes

1. Much work exists in various fields to describe what semantics is. Unfortunately, there is no consensus. We will limit our attention to the definitions that are relevant to our modeling needs.
2. The definition of *roleof* is similar to that of *categoryof* [6].
3. SWITCH is a trademark of Bellcore. This schema should be seen as a realistic schema. No implication should be made about sufficiency or accuracy with respect to the real system.
4. If there are k classes $a_1, a_2, \ldots, a_k$ that have the same object type A, then the same ideas apply, making use of k mappings.

References

1. A. Aho, J. Hopcroft, and J. Ullman, "Data structures and algorithms," Addison Wesley: Reading, MA, 1987.
2. C. Batini, M. Lenzerini, and S. Navathe, "A comparative analysis of methodologies for database schema integration," ACM Computing Surveys, vol. 18, no. 4, pp. 323–364, December 1986.
3. M. Brodie, "On the development of data models," On Conceptual Modeling, Springer Verlag, 1984.
4. M.L. Brodie, and F. Manola, *Database Management: A Survey.* Readings in Artificial Intelligence and Databases, J. Mylopoulos and M.L. Brodie (Eds.), Morgan Kaufmann Publishers: San Mateo, CA, 1989.
5. U. Dayal, and H. Hwang, "View definition and generalization for database integration in a multidatabase system." *IEEE Trans. on Soft. Eng.*, SE-10, 6, (Nov.), pp. 628–644, 1984.
6. R. Elmasri, and S. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings: Redwood City, CA, 1989.
7. D. Fishman, et al., "IRIS: An object-oriented DBMS," *ACM Trans. on Office Info. Syst.*, vol. 4, no. 2, April 1987.
8. J. Geller, A. Mehta, Y. Perl, E. Neuhold, and A. Sheth, Algorithms for Structural Schema Integration, *Second International Conference for Systems Integration*, Morristown, NJ, pp. 604–614, 1992.
9. J. Geller, Y. Perl, P. Cannata, A. Sheth, and E. Neuhold, "A case study of structural integration," *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD, pp. 102–111, November 1992.
10. J. Geller, Y. Perl, and E.J. Neuhold, "Structure and semantics in OODB class specifications," SIGMOD RECORD, vol. 20, no. 4, pp. 40–43, 1991.
11. J. Geller, Y. Perl, E.J. Neuhold, and A. Sheth, "Structural schema integration with full and partial correspondence using the Dual Model," *Information Systems*, vol. 17, no. 6, pp. 443–464, 1992.
12. M. Gyssens, J. Paredaens, and D.V. Gucht, "A graph-oriented object model for database end-users interfaces," Proceedings of the ACM SIGMOD Conference, Atlantic City, pp. 24–33, 1990.
13. A. Goldberg, and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley: Reading, MA, 1983.
14. M. Halper, J. Geller, Y. Perl, and E. Neuhold, "A graphical schema representation for object-oriented databases," *First International Workshop on Interfaces to Database Systems (IDS-92)*, Glasgow, Scotland, 1992 (to be published by Springer Verlag).
15. W. Kim, "Research direction for integrating heterogeneous databases," *1989 Workshop on Heterogeneous Databases*, Chicago, IL, December 1989.

16. W. Klas, "A metaclass system for open object-oriented data models," *Ph.D. dissertation*, Technical University of Vienna, Austria, 1990.
17. W. Klas, E.J. Neuhold, R. Bahlke, K. Drosten, P. Fankhauser, M. Kaul, P. Muth, M. Oheimer, T. Rakow, and V. Turau, "VML design specification document," *Tech Report, GMD-IPSI*, Germany, 1992.
18. W. Klas, E.J. Neuhold, and M. Schrefl, "On an object-oriented data model for a knowledge base," In *Research into Networks and Distributed Applications—EUTECO 88*, R. Speth (Ed.), North-Holland, 1988.
19. C. Lecluse, and P. Richard, "Modeling inheritance and genericity in object-oriented databases," *LNCS #326*, ICOT, Japan, pp. 223-237, 1988.
20. E.J. Neuhold, J. Geller, Y. Perl, and V. Turau, "A theoretical underlying Dual Model for knowledge-based systems," *Proc. of the First Intl. Conf. on Systems Integration*, Morristown, NJ, pp. 96–103, 1990.
21. E.J. Neuhold, Y. Perl, J. Geller, and V. Turau, "Separating structural and semantic elements in object-oriented knowledge bases," *Proc. of the Advanced Database System Symposium*, Kyoto, Japan, pp. 67-74, 1989.
22. E.J. Neuhold, Y. Perl, J. Geller, and V. Turau, "The Dual Model for object-oriented data bases," New Jersey Inst. of Tech., *Tech Report CIS-91-30*, 1991.
23. F. Saltor, "On the power to derive external schemata form the database schema," *Proc. of the 2nd IEEE Int. Conf. on Data Engineering*, Los Angeles, CA, 1986.
24. A. Sheth, "Issues in schema integration: Perspective of an industrial researcher," presented at *ARO Workshop on Heterogeneous Databases*, Philadelphia, PA, September 1991.
25. A. Sheth, and J. Larson, "Federated database systems for managing distributed heterogeneous, and autonomous databases," *ACM Computing Surveys*, pp. 183-236, September 1990.
26. G.L. Steele, Jr., *Common LISP—The Language*, Second Edition, Digital Press: Bedford, MA, 1990.