Information Systems Vol. 17, No. 6, pp. 443–464, 1992
Printed in Great Britain. All rights reserved

0306-4379/92 $5.00 + 0.00
Copyright © 1992 Pergamon Press Ltd

STRUCTURAL SCHEMA INTEGRATION WITH FULL AND PARTIAL CORRESPONDENCE USING THE DUAL MODEL

JAMES GELLER,[1] YEHOSHUA PERL,[1] ERICH NEUHOLD[2] and AMIT SHETH[3]

[1]Institute for Integrated Systems, CIS Department and Center for Manufacturing Systems, New Jersey Institute of Technology, Newark, NJ 07102, U.S.A.

[2]Institute for Integrated Publication and Information Systems, GMD, Darmstadt, Germany

[3]Bellcore, 444 Hoes Lane, Piscataway, NJ 08854, U.S.A.

(Received 10 July 1991; in final revised form 6 August 1992)

Abstract—The integration of views and schemas is an important part of database design and evolution and permits the sharing of data across complex applications. The view and schema integration methodologies used to date are driven purely by semantic considerations, and allow integration of objects only if that is valid from both semantic and structural view points. We discuss a new integration method called structural integration that has the advantage of being able to integrate objects that have structural similarities, even if they differ semantically. This is possible by using the object-oriented Dual Model which allows separate representation of structure and semantics. Structural integration has several advantages, including the identification of shared common structures that is important for sharing of data and methods.

Key words: Object-oriented databases, database integration, schema integration, view integration, structure and semantics, Dual Model, structural integration

1. INTRODUCTION

In most enterprises, multiple database systems have been acquired or built over the last several years. With the advent of easy data communication and powerful workstations many applications in these enterprises are now being extended to interoperate and to utilize several of these databases. However, as with other aspects of systems integration [1, 2] the process of accessing and manipulating multiple databases should not be left to the application alone.

An important approach to database integration was introduced in the form of *federated databases* by [3] and later on by [4]. Sheth and Larson [5] present an autonomy-oriented taxonomy of federated databases, in which they distinguish between loosely coupled systems (e.g. [6]) and tightly coupled systems (e.g. [7]). Many approaches to database integration rely on traditional data models, possibly with some extensions; e.g. [8, 9] use extended forms of the ER model. In contrast, it is believed by a number of researchers that object-oriented databases will aid in the problem of integrating heterogeneous components (e.g. [10, 11]). Approaches towards object-oriented integration methodologies have been reported, e.g. in [12, 13, 14].

For our purpose we want to restrict the scenario to two aspects of integration.

1. Multiple autonomous databases will have to cooperate with respect to data redundancy, data integration, data consistency and data exchange. For expressing these aspects we have to build in a *bottom-up* fashion database views that integrate the different database schemas and make them available in a homogenized (symbolic or semantic) fashion to the application programs.
2. In multiple applications, each will require its own subset (view) of the data from the underlying database. On the other hand, many of these data will be shared between cooperating applications. In order to achieve proper behavior in this sharing we will have to integrate conceptually in a *top-down* fashion at least some of these application data into a common (view) schema of the database.

We may map these two scenarios into the five level architecture proposed in [5] and illustrated in Fig. 1. The bottom-up approach essentially requires that the (local) schema of a participating database is at least partially transformed into the modeling language used for the multi-database

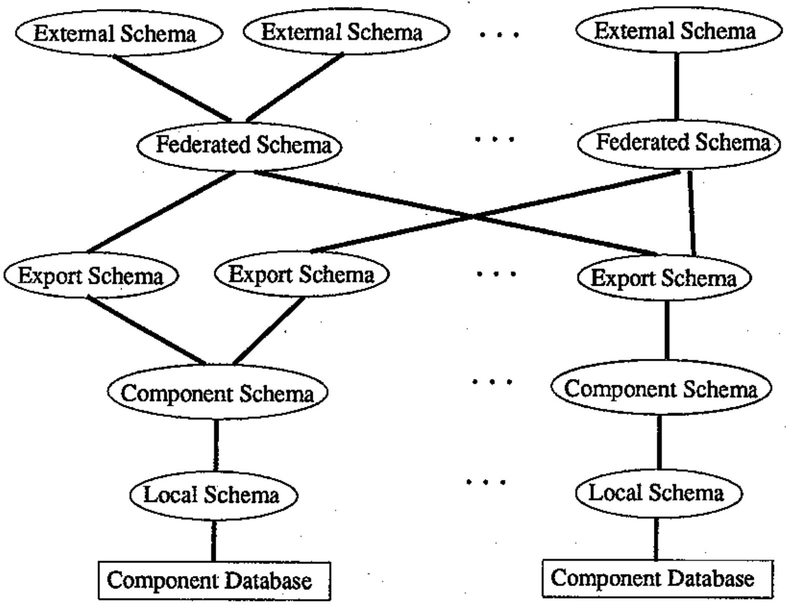

Fig. 1. Five level schema architecture (from [5]).

system. From this compound schema, subsets will be created as *export schemas* to be integrated into one (or many) common (federated) schema. In this paper we shall use the term *database integration* when we discuss the bottom-up approach.

In the top-down approach the external schemas of cooperating applications will have to be combined into the common schema. In this paper we will use the term *view integration* when we describe the top-down process of combining the database needs of different cooperating applications into a common schema. As a common name for both operations, we will use the term *schema integration.*

Of course, in general the top-down approach and the bottom-up approach will be utilized together to arrive at proper common schemas. In addition it should be clear that for each of the external and common schemas additional data (schema information and instance values) that cannot be placed into the autonomous component databases will have to be kept. Different claims have been made by different systems on whether there should exist a single central database system, a single distributed database system or multiple database systems for this purpose. In PEGASUS [15] the choice has been made for using as the default database system the database of the individual user (group). In VODAK [16] a distributed homogeneous object-oriented database system is used.

When multiple databases are developed independently, it is unavoidable that certain incompatibilities will make the integration process difficult. As has been pointed out in [14], naming differences, structural differences and semantic differences may occur between the different databases. In [17] it was shown that similar difficulties arise in view integration for cooperating applications. Already in early integration work, common schemas were not expressed in a relational database language but, for instance, in the semantically richer model DAPLEX/ADAPLEX [18]. Similarly, many later approaches were either based on ER models [8] or object-oriented data models [12]. Of course, if the relational data models would be expanded as indicated in the Third Generation Database Manifesto [19] such a rich relational model could well form the semantic integration tool required here. The process of integration in all of these models is based on one hand on semantic generalization principles derived from Artificial Intelligence and on the other hand on principles of structural inheritance derived from object-oriented programming languages.

In this paper we will show how a gain in expressive power beyond the above modeling tools can lead to better structural and semantic integration of the different schemas and schema components. We shall use the Dual Model [20, 21, 22, 23] that separates *structural* and *semantic* features of a database schema. (Different data models assign different meanings to the terms structure and semantics, and we will later clarify the distinctions made by the Dual Model.)

In the Dual Model we define for each class an object type. The object type captures the structural properties of a class. The semantic properties are reflected in the class specification. One can say that the structural properties have been extracted from the class specification, yielding the corresponding object type specification. When two classes are similar in their structure, we can use the same object type for both of them. In the body of this paper more details of the Dual Model will be supplied, whenever necessary.

An interesting effect of the separation of structure and semantics in the Dual Model is that four possibilities exist with respect to similarity between two database schemas.

1. Two schemas may be semantically as well as structurally similar. In this case, standard generalization-based integration tools can be used successfully.
2. Two schemas may be semantically as well as structurally different. In this case, no integration is possible.
3. Two schemas may be semantically different but structurally similar. In this case, generalization-based integration fails; however, structural integration, the topic of this paper, is possible. This case is of considerable interest for the top-down process of (view) integration.
4. Finally, there is the mirror case of the previous case, where two schemas may be semantically similar but structurally different. While this case is of considerable interest in bottom-up (database) integration, we have to defer its discussion to future work.

Structural integration provides the following significant advantages: (a) extended range of solvable integration problems; (b) simplified specification of schemas; (c) code sharing; (d) simplified understanding of schemas; and (e) reusable query optimization.

Extended range of solvable integration problems: structural integration permits in many cases to integrate classes which cannot be integrated with standard generalization-based techniques. This advantage has been the motivating factor for the inception of structural integration as a technique.

Simplified specification: the structural integration process allows sharing of the specification of properties (e.g. attributes). The shared type definitions act as templates that can be reused to define multiple classes that are structurally similar (and hence share a type definition). This advantage is similar to the advantage conventionally achieved by integration by generalization.

Code sharing: among the properties that may be shared are also methods which are central elements of object-oriented class specifications. Therefore, structural integration amounts to a kind of code-reusability.

Simplified understanding of schemas: this is a cognitive advantage for a user that is faced with the need to understand two or more schemas. The result of structural integration is a new schema that captures the important connectivity features of the schemas it integrates. This integrated schema is always smaller than the sum of the sizes of the schemas it integrates, and ideally is comparable in size to the largest of the integrated schemas. The user can therefore get a quick understanding of the database by first studying the integrated schema and then applying his understanding to the two or more original schemas.

Reusable query optimization: much of DBMS query optimization is based on syntactic information. A query optimization strategy can be developed first for a structural subschema, and then it can be applied to the semantic schemas that share the structural subschema.

This paper is the second in a sequence of two papers introducing structural integration. The first paper [24] is application-oriented and demonstrates the occurrence of structural integration in an actual industrial database. The theory in [24] is limited to full structural correspondence and a special case of partial correspondence which were found to be sufficient for almost all problems of the application. In this paper, we provide the complete theoretical framework of full and partial structural correspondence.

Section 2 supplies additional material necessary for demonstrating structural integration. An example domain is introduced in Subsection 2.1. Subsection 2.2 discusses details of the Dual Model and of the textual representation of the Dual Model. A graphical notation for the Dual Model will be presented in Subsection 2.3. Using this graphical notation, the example domain will be presented formally.

Section 3 discusses structural integration. Subsection 3.1 formally treats the mappings between object types and classes which are a central issue of the Dual Model and play an important role in structural integration. Subsection 3.2 formally defines full structural correspondence and partial structural correspondence and discusses integration for both these cases. In Section 4 we discuss the question of how one could decide which classes are possible candidates for structural integration. Section 5 contains our conclusions. A preliminary short version of this paper appeared in [25].

2. THE DUAL MODEL REPRESENTATION OF TWO DATABASES

2.1. *Purchasing department and inventory control*

In order to have a concrete example available, an application domain is now introduced consisting of a purchasing department database and an inventory control database. In our very simplified model, the activity of the purchasing department is to issue a purchase order to a supplier. The department therefore needs a database containing information about all purchase orders, items to be purchased and suppliers.

The inventory control department issues receiving reports. Its database contains information about items, receiving reports and suppliers. However, the items that are received are not necessarily the same as those which were ordered, because suppliers sometimes deliver an item that they consider as equivalent, e.g. a new model of a device that replaces an older model.

To demonstrate the process of schema integration, two separate object-oriented databases, one for the purchasing department and one for the inventory control, will be introduced in this section and integrated in Section 3. Before that, a few general comments about the Dual Model and its graphical notation will be made in the next two subsections.

2.2. *The Dual Model and its notation*

The characteristics of object-oriented database systems are, among others, the notions of objects and classes. A class can be regarded as a container for objects which are similar in their structure and their semantics in the application.

To describe the structure and semantics of objects, the class uses the following properties.

1. Attributes—contain printable values of a given data type.†
2. Relationships—contain pointers to other classes. Relationships are user defined.
3. Generic relations—contain pointers to other classes. They differ from relationships in that they are predefined in the system. Such relations are for instance *roleof* and *setof*.
4. Methods—specify operations which can be applied to instances of a given class. A method is either an *access path method* or a segment of code in the host language. The second type is referred to as an *operation* and is usually not shown in the database schema. In this paper we concentrate on access path methods.

   An *access path method* is a chain of relationships or generic relations. This chain may be terminated by an attribute. It permits the retrieval of an attribute that may be relevant to an instance of a class, but that is stored with a different class. An access path method may be visualized as follows:

   $$class_1 \xrightarrow{Relationship_1} class_2 \xrightarrow{Relationship_2} class_3 \xrightarrow{Relationship_3} \cdots \xrightarrow{Relationship_{n-1}} class_n \xrightarrow{Attribute} result.$$

   In the path from $class_1$ to $class_n$ the relationship $Relationship_k$ is said to be at position k for $1 \leqslant k < n$. For a complete and formal definition of access path methods see [21].

In many systems, (e.g. ONTOS [26], ObjectStore [26], O_2 [26, 27] and others [28, 29, 30, 31, 32]) the subclass hierarchy is used for two purposes at the same time: (1) to factorize common structure and behavior of classes and (2) to express additional semantic relations between classes. This leads to a situation that two classes modeling semantically related objects can only be represented correctly, if the objects in question are structurally related as well.

In the Dual Model structural information and semantic information are described separately. This results in improved modeling capabilities. To clarify this claim, it is necessary to summarize the definitions of structure and semantics in the Dual Model [22].

†Some systems use the term properties limited to what we refer to as attributes. This explicit definition will hopefully help to avoid confusion.

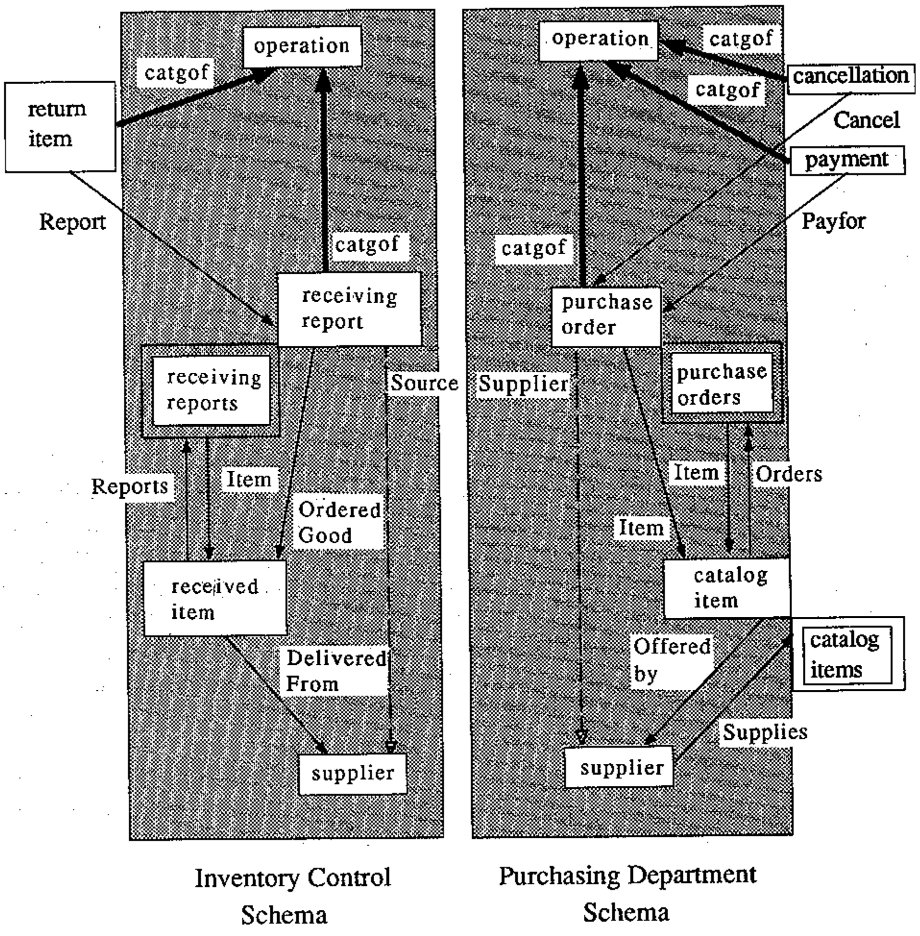

Fig. 2. Two subschemas of the application domain.

An 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. 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 properly describes the connection between the corresponding real world objects and their features. Note that the names of a property are considered semantic in other models (e.g. ER models) but are not considered semantic in the Dual Model.

As just mentioned, structural and semantic information are represented separately. Structural information is represented in an *object type*. Semantic information is represented in the corresponding *object class* (or, in short, *class*). Every class has a unique associated object type, but one object type can function as the object type for several structurally identical classes.

An object class in the Dual Model is either specified by code or by a graphical notation. The graphical notation will be introduced in Subsection 2.3, but the reader may want to use Fig. 2 already at this point. In a code specification, the term **class** precedes the name of an object class. The properties follow in this order: (1) generic relations, (2) attributes, (3) relationships and (4) methods. Keywords, such as **attributes**, separate the different property sections. Property names (except for generic relations) have the first letter capitalized. Class names appear in small letters. As an example, we consider the class purchase_order from our application domain:

class purchase_order
  categoryof: operation
  memberof: purchase_orders
  attributes:

```
      OrderNumber: INTEGER
      Quantity: INTEGER
      Unit: STRING
      Cost: DOLLARTYPE
      OrderDate: DATETYPE
      OrderingDepartment: STRING
    essential: OrderNumber
    relationships:
      Item: catalog_item
    methods:
      Supplier( ):
      Item → catalog_item:
      OfferedBy → supplier
```

The attributes of a purchase_order are OrderNumber, Quantity, Unit, Cost, OrderDate and OrderingDepartment. There is a single relationship Item and a single method Supplier.

The Dual Model contains two kinds of specialization generic relations between classes. The first one is *categoryof* which relates the specialized class to the more general class when both are in the same context. The second is *roleof* which relates the specialized class to the more general class when the two are in different contexts. The term *context* describes semantic information since it cannot be decided mathematically whether two entities occur in the same context. Thus, *roleof* and *categoryof* are semantic generic relations. The above class purchase_order is *categoryof* the class operation. Therefore, a purchase_order is an operation, and both purchase_order and operation occur in the same context.

Whenever we need a relationship to refer to a set of objects, we can define a class to represent this set. The connection between the set class and the member class is expressed with the *setof* and *memberof* generic relations. We now present the class purchase_orders which is the set class of the class purchase_order.

```
  class purchase_orders
    setof: purchase_order
    attributes:
      NumberofOrders: INTEGER
      GroupPurpose: STRING
    relationships:
      Item: catalog_item
```

The class purchase_orders refers with a *setof* to purchase_order, and the class purchase_order refers with a *memberof* to purchase_orders. So far we have only shown the "class-side" of the Dual Model. We now present an object type, namely FORM.

```
  objecttype FORM
    subtypeof: OPERATION
    memberof: FORMS
    attributes:
      OrderNumber: INTEGER
      Quantity: INTEGER
      Unit: STRING
      Cost: DOLLARTYPE
      FormDate: DATETYPE
      OrderingDepartment: STRING
    relationships:
      Item: ITEM
    methods:
      Supplier( ):
      Item → ITEM:
      OfferedBy → SUPPLIER
```

Object types look similar to classes, but they are preceded by the term **objecttype** and their names are capitalized. A *subtypeof* generic relations connects a refined object type to a more general object type, enabling inheritance of the properties of the general object type to the refined object type. The object type FORM is a *subtypeof* the object type OPERATION (not shown) and as such inherits, e.g. all the attributes from OPERATION.

In [21] we show that if a class has a *categoryof* relation, then the corresponding object type has a *subtypeof* relation, while for a *roleof* relation this may or may not be the case. Thus, whenever we specify a class for which no object type is given explicitly and which has a *roleof* that corresponds to a *subtypeof*, we add to the *roleof* specification an additional (*subtypeof*) annotation.

To summarize the specialization relations of the Dual Model, *categoryof* and *roleof* are semantic (occur in classes only) while *subtypeof* is structural (occurs in object types only). The *setof* and *memberof* generic relations were previously introduced for classes, but corresponding structural generic relations exist also. For instance, the object type FORM is *memberof* the object type FORMS.

As was seen in the two code examples, both object types and classes contain the same kinds of properties (attributes, relationships, methods, generic relations), however, with different interpretations. The specification of properties for both object types and classes might appear burdensome, but it is not always done explicitly. When there is only one class for a given object type, only the class needs to be specified, and the object type is defined implicitly through defaults. In a schema using the Dual Model, an object type is only declared explicitly when there are several classes corresponding to it. The specification of these classes follows the specification of the object type and describes the semantic aspects of each class.

The representation of a class may contain two kinds of semantic constraints. The first constraint is that of *essential* properties. The existence of an object is conditioned on the existence of its essential properties. An instance of a class can only exist if the values of its essential properties are different from NIL. In the class purchase_order, there is an essential attribute OrderNumber.

The second constraint is that of a *dependent relationship*. If the existence of an object depends on the existence of another object, we can model this with a dependent relationship. Suppose an object has several dependent relationships (meaning that several objects are dependent on the existence of this object), then the deletion of the object has the consequence that the dependent objects are also deleted. The semantic constraints do not play a role in structural integration. They occur in the class but not in the object type. This permits the integration of two classes that differ in their semantic constraints, by one common object type.

The previously shown classes purchase_order and purchase_orders, as well as the following class catalog_item form the backbone of the schema of the purchasing department.

```
class catalog_item
  memberof: catalog_items
  attributes:
    CatalogNumber: INTEGER
    Name: STRING
    Price: DOLLARTYPE
    Description: STRING
  essential: CatalogNumber
  relationships:
    OfferedBy: supplier
    (dependent) Orders: purchase_orders
```

The relationship Orders (in catalog_item) to purchase_orders has to be modeled as a dependent relationship since you cannot order a nonexistent item. An instance of the previous class purchase_orders represents all the purchase orders issued for a specific instance of catalog_item, referenced by the relationship Item (in purchase_orders).

The class purchase_order also contains an access path method, Supplier. Access path methods defined in classes are constructed from pairs of a property and the class that it refers to. Arrows ($\rightarrow$) separate pair elements, while colons (:) separate consecutive pairs.

The access path method Supplier retrieves the supplier of a given order. It uses the relationship Item to retrieve the correct catalog_item and then the relationship OfferedBy, defined in catalog_item, to retrive the supplier. The Supplier method eliminates the need for explicitly maintaining a relationship Supplier in the class purchase_order and at the same time guarantees that the supplier referred to in the order is actually the supplier of the catalog_item.

The object type FORM contains a corresponding structural method, also called Supplier. In a structural method, pairs consist of properties and object types. Object types will be introduced in more detail as integration tools in Subsection 3.3.

A complete example of Dual Model code will be shown in the Appendix only. It is necessary to supply an interface connecting the corresponding object type and object class(es). This interface is realized by two features. The first one is that for each class we have a declaration of its object type. This is expressed by the **objecttype** keyword in the class definition. The second is a mapping M from the set P of properties of the object type to the set Q of properties of the class. This mapping is a one-to-one function from P onto Q. It identifies for each property of the object type the corresponding property of the class. This mapping will be discussed in detail in Section 3.

We conclude this section by showing the classes of the inventory control database that will be integrated with the already introduced classes of the purchasing department. The integration will be performed by finding common object types for pairs of classes.

Inventory control

```
class receiving_report
  categoryof: operation
  memberof: receiving_reports
  attributes:
    OrdNum: INTEGER
    DeliveryDate: DATETYPE
    Quantity: INTEGER
    UnitofMeasurement: STRING
    ReceivingDepartment: STRING
    Cost: DOLLARTYPE
  essential: OrdNum
  relationships:
    OrderedGood: received_item
  methods:
    Source( ):
    OrderedGood → received_item:
    DeliveredFrom → supplier
```

Note that each receiving report is limited in our simplified example to a given quantity of only one item. (The same was true for the previous class purchase_order.) The method Source has a similar structure to the method Supplier in the class purchase_order. An instance of the following class receiving_reports represents all the reports for a specific instance of received_item. However, deleting an old instance of received_item does not necessarily imply the deletion of a receiving_report of this item. Thus the Reports relationship in received_item is not dependent, as opposed to the Orders relationship in catalog_item.

```
class receiving_reports
  setof: receiving_report
  attributes:
    Quantity: INTEGER
    GroupPurpose: STRING
  relationships:
    Item: received_item

class received_item
  attributes:
    CatalogNumber: INTEGER
    Price: DOLLARTYPE
    Name: STRING
    Damaged: BOOLEAN
  relationships:
    DeliveredFrom: supplier
    Reports: receiving_reports
```

2.3. Graphical notation

The details of the two departments are explained using a graphical notation for object-oriented databases that was introduced in [33]. Classes are represented as boxes. Heavy line arrows represent generic relations such as *roleof* and *categoryof*. Thin arrows represent relationships between classes. A box with a double frame represents a set class. It is drawn to share one corner with the box that represents its member class. Dependent relationships are marked with a double-headed arrow. An access path method is shown as a dotted thin arrow from the class where it is defined to the class where the path ends.

For example, in Fig. 2 (right side), which represents the purchasing department, the box purchase_order represents the class of all purchase orders. It shares a corner with the box purchase_orders which is the class of all sets of purchase orders. Every purchase_order is *categoryof* an operation, which means that both classes occur in the same real-world context. The class catalog_item has a dependent relationship Orders to the class purchase_order. The class purchase_order has an access path method Supplier that ends at the class supplier. Figure 2 (left side) represents the inventory control.

The details and exact conventions as well as the motivations of the graphical representation language can be found in [33], but the information given in this subsection should be sufficient for getting an understanding of the domain. Both sides of Fig. 2 show only small subsets of realistic schemas. Parts of these subsets were previously introduced as code examples in Subsection 2.2.

3. STRUCTURAL INTEGRATION

3.1. The mapping between types and classes

As pointed out previously, it is necessary to supply an interface connecting the corresponding object type and object class(es). This interface is implemented by two features. The first one is that for each class we have a declaration **objecttype:** in the class definition. The second is a mapping M from the set of P of properties of the object type to the set Q of properties of the class. This mapping is a one-to-one function from P onto Q and was first introduced in [34]. It identifies for each property of the object type the corresponding property of the class. In many cases, an object type A has only one class a associated. In that case, we implicitly define the object type A so that the mapping satisfies a set of conditions that will be described after introducing the following notational conventions.

Every property can be viewed as a pair, consisting of the name of the property and the name of the type (or class) this property is referring to. For instance, in the previously shown class received_item, the property Price could be viewed as the pair (Price, DOLLARTYPE).

The function "selector" returns the first element of any pair.

$$\text{selector}((u, v)) = u.$$

For simplicity we will omit the inner pair of parentheses.

$$\text{selector}(u, v) = u.$$

For example,

$$\text{selector}((\text{Price, DOLLARTYPE})) = \text{selector}(\text{Price, DOLLARTYPE}) = \text{Price}.$$

The function "datatype" expects as argument a pair that describes an attribute. It returns the second element of the pair. It is undefined ($\perp$) for any other kind of argument.

$$\text{datatype}(u, v) = \begin{cases} v & \text{if}(u, v) \text{ describes an attribute} \\ \perp & \text{otherwise} \end{cases}$$

The following functions are similar:

$$\text{class}(u, v) = \begin{cases} v & \text{if } (u, v) \text{ describes a relationship or generic relation defined in a class} \\ \perp & \text{otherwise} \end{cases}$$

$$\text{objecttype}(u, v) = \begin{cases} v & \text{if } (u, v) \text{ describes a relationship or generic relation defined in an object type} \\ \perp & \text{otherwise} \end{cases}$$

$$\text{relationname}(u, v) = \begin{cases} u & \text{if } (u, v) \text{ describes a generic relation} \\ \perp & \text{otherwise} \end{cases}$$

We now enumerate the conditions for the existence of a mapping between an object type and an object class. This mapping provides the interface from an object type to the corresponding class. This interface enables the specification of the actual classes referred to implicitly in the object type description. An understanding of these conditions is necessary for understanding the algorithm (Subsection 3.2.1) which constructs an object type that structurally integrates two or more classes.

Conditions for mapping M from object type to one class

The conditions for the existence of a mapping M from an object type A to a class a are:

1. The corresponding properties of the object type and the class have the same selector.
2. Corresponding attributes have the same data type.
3. For every relationship r defined in an object type A and pointing to an object type B, there must exist a corresponding relationship r defined in the class a. This relationship r of the class a refers to a class b that must have an object type B.
4. For every *subtypeof* generic relation r from an object type A to an object type B, the class a has either a *categoryof* or a *roleof*† relation to a class b having an object type B, but there might be no connection at all.
5. For every *memberof* (*setof*) generic relation r from an object type A to an object type B, the corresponding r is a *memberof* (*setof*) relation from the class a to a class b having the object type B.
6. For every access path method m (as defined at the beginning of Subsection 2.2) from an object type A referring to a sequence of object types $B_1, B_2, \ldots, B_k$, the access path method m of the class a refers to a sequence of classes $b_1, b_2, \ldots, b_k$ such that the class b_i, $1 \leqslant i \leqslant k$, has the object type B_i, $1 \leqslant i \leqslant k$. Furthermore, let r_i be the relationship or relation of B_i such that $r_i(B_i) = B_{i+1}$, $1 \leqslant i < k$, and let s_i be the relationship or generic relation of b_i such that $s_i(b_i) = b_{i+1}$, $1 \leqslant i < k$, then the mapping from the properties of B_i to the properties of b_i must satisfy a mapping $M'(r_i) = s_i$. This mapping M' is identical to M only for $k = 1$. For all other generic relations (relationships) in the path, M' is dependent on the class in which the generic relation (relationship) is defined.

Consider now the case where two classes a_1 and a_2 have the same object type A. 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 $q_2 \in Q_2$. If there are k classes $a_1, a_2, \ldots, a_k$ with the same object type then the same ideas apply, making use of k mappings.

Conditions for mappings from object type to several classes

1. 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.

†With a *subtypeof* annotation

2. The selectors of the properties p, q_1 and q_2 are not necessarily identical; however, if selector(q_1) = selector(q_2), then selector(p) = selector(q_1).
3. Let p, q_1 and q_2 be attributes. Then datatype(p) = datatype(q_1) = datatype(q_2).
4. Let p be a relationship from an object type A to an object type B, q_1 be a relationship from a class a_1 to a class b_1 and q_2 be a relationship from a class a_2 to a class b_2. Then b_1 and b_2 should have the same object type B.
5. Let p be a *subtypeof* relation from an object type A to an object type B. Then the relations q_1 from a_1 and q_2 from a_2 are either *categoryof* or *roleof* relations to the classes b_1 and b_2, respectively, such that b_1 and b_2 have the same object type B, or q_1 and q_2 might be undefined.
6. Let p be a *memberof* or a *setof* relation from an object type A to an object type B. Then the relations q_1 from a_1 and q_2 from a_2 are *memberof* or *setof* relations to the classes b_1 and b_2, respectively, such that b_1 and b_2 have the same object type B.
7. Let p be any relationship or generic relation at position i in an access path method from an object type A to an object type B, so that p effectively connects two object types K and L. Then q_1 connects two classes k_1 and l_1, and q_2 connects two classes k_2 and l_2 such that k_1 and k_2 have the object type K and l_1 and l_2 have the object type L, and both q_1 and q_2 are at position i in their respective access path methods.

3.2. Formal conditions for structural integration

Consider two sets of classes $\mathbf{C} = \{a_1, a_2, \ldots, a_n\}$ and $\mathbf{D} = \{b_1, b_2, \ldots, b_n\}$ of equal cardinality $|\mathbf{C}| = |\mathbf{D}|$, i.e. both sets contain the same number of classes. An example of such a pair of sets are the classes in the grey block of Fig. 2 (right side) within the PURCHASING_DEPARTMENT schema and the classes in the grey block within the INVENTORY_CONTROL schema in Fig. 2 (left side). Structural integration between the sets **C** and **D** is possible if there exists a correspondence between **C** and **D** such that for every two corresponding classes $a \in \mathbf{C}$ and $b \in \mathbf{D}$, one can construct a common object type. There are two cases of correspondence, full structural correspondence (Subsection 3.2.1) and partial structural correspondence (Subsection 3.2.2).

3.2.1. Integration of classes with full correspondence. In order to construct a common object type for two corresponding classes a and b, there must exist a full structural correspondence between these two classes, i.e. between their sets of properties. Full correspondence for attributes means that their data types must be identical. Full correspondence for relationships means that the referenced 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 identical and refer to classes of the same object type. (For exceptions see below.)

Formally, let the class $a(b)$ have a set $\{x_i\}(\{y_i\})$ of attributes, a set $\{r_i\}(\{s_i\})$ of relationships, a set $\{m_i\}(\{n_i\})$ of access path methods and a set $\{g_i\}(\{h_i\})$ of generic relations to other classes. The classes a and b stand in *full structural correspondence* if and only if:

1. There exists a one-to-one correspondence between the sets of attributes $\{x_i\}$ and $\{y_i\}$ such that if x_i corresponds to y_i, then datatype(x_i) = datatype(y_i).
2. There exists a one-to-one correspondence between the sets of relationships $\{r_i\}$ and $\{s_i\}$ such that it must be possible to construct a common object type for class(r_i) and class(s_i).
3. There is a one-to-one correspondence between the sets of generic relations $\{g_i\}$ and $\{h_i\}$ such that (1) relationname(g_i) = relationname(h_i) or both relation names are members of the set $\{$*roleof*, *categoryof*$\}$; and (2) it must be possible to construct a common object type for class(g_i) and class(h_i).
4. There exists a one-to-one correspondence between the sets of access path methods $\{m_i\}$ and $\{n_i\}$ such that if m_i is a method that defines a path going through the sequence $a_1, a_2, \ldots, a_s$ of classes, and n_i defines a similar path $b_1, b_2, \ldots, b_t$ of classes, then the following conditions hold: (a) $s = t$ and (b) it must be possible to construct a common object type for a_i and b_i. Furthermore, consider the case that a_i and b_i have the same object type, say A_i. Let a_{i+1} and b_{i+1} have the same object type, say A_{i+1}; and let $p_i(q_i)$ be the property connecting a_i to a_{i+1} (b_i to b_{i+1}). Let P_i be the property connecting A_i to A_{i+1}. Then the mappings M_1' and M_2' from A_i to the classes a_i and b_i, respectively, satisfy $M_1'(P_i) = p_i$ and $M_2'(P_i) = q_i$.

Some more notational conventions are needed at this point. The function "typeof" returns the object type A of a class a.

$$\text{typeof}(a) = A.$$

The function *typeof* (in italics!) returns the object type A of the class a of a property $p = (x, a)$ in pair notation.

$$\textit{typeof}(p) = \text{typeof}(\text{class}(p)) = \text{typeof}(a) = A.$$

The function "class-sequence" returns the sequence of all classes that occur in an access path method m defined in a class.

$$\text{class-sequence}(m) = \langle a_1, a_2, \ldots, a_n \rangle.$$

Similarly, "type-sequence" returns the sequence of all object types that occur in an access path method m defined in an object type.

$$\text{type-sequence}(m) = \langle A_1, A_2, \ldots, A_n \rangle.$$

We now present an algorithm for creating a common object type for two classes with full structural correspondence.

Structural_integration (in a, b: class; out A: object type)

1. For two corresponding attributes x_i, y_i with selector(x_i) = selector(y_i) define in the object type A an attribute (selector(x_i), datatype(x_i)).
2. For two corresponding attributes x_i, y_i with selector(x_i) $\neq$ selector(y_i) define in A an attribute (z, datatype(x_i)) with z = selector(x_i), or z = selector(y_i), or z is a freely chosen new selector.
3. For two corresponding relationships r_i, s_i with selector(r_i) = selector(s_i) define in A a relationship (selector(r_i), *typeof*(r_i)).
4. For two corresponding relationships r_i, s_i with selector(r_i) $\neq$ selector(s_i) define in A a relationship (z, *typeof*(r_i)) with z = selector(r_i), or z = selector(s_i), or z is a freely chosen new selector.
5. For two corresponding *memberof* or *setof* generic relations, g_i and h_i, define in A a generic relation (selector(g_i), *typeof*(g_i)). Note that by the definition of full structural correspondence it must be the case that selector(g_i) = selector(h_i).
6. For two corresponding *categoryof* or *roleof*† generic relations, g_i and h_i, define in A a generic relation (subtypeof, *typeof*(g_i)).
7. For two corresponding methods, m_i, n_i with selector(m_i) = selector(n_i) and class-sequence(m_i) = $\langle k_1, k_2, \ldots, k_s \rangle$ define in A a method m, such that selector(m) = selector(m_i) and type-sequence(m) = $\langle$typeof(k_1), typeof(k_2), ..., typeof(k_s)$\rangle$.
8. For two corresponding methods, m_i, n_i with selector(m_i) $\neq$ selector(n_i) and class-sequence(m_i) = $\langle k_1, k_2, \ldots, k_s \rangle$ define in A a method m, such that selector(m) = z, and type-sequence(m) = $\langle$typeof(k_1), type(k_2), ..., typeof(k_s)$\rangle$. Like before, z = selector(m_i), or z = selector(n_i), or z is a newly introduced selector.

As an example of full structural correspondence, we remind the reader of the classes purchase_order from the purchasing department database, and receiving_report from the inventory control database, all of which were previously shown. (We repeat the code for readability.)

```
class purchase_order
  categoryof: operation
  memberof: purchase_orders
  attributes:
    OrderNumber: INTEGER
    Quantity: INTEGER
    Unit: STRING
    Cost: DOLLARTYPE
    OrderDate: DATETYPE
    OrderingDepartment: STRING
```

†We assume that every *roleof* has an associated *subtypeof*.

```
  essential: OrderNumber
  relationships:
    Item: catalog_item
  methods:
    Supplier( ):
    Item → catalog_item:
    OfferedBy → supplier

class receiving_report
  categoryof: operation
  memberof: receiving_reports
  attributes:
    OrdNum: INTEGER
    DeliveryDate: DATETYPE
    Quantity: INTEGER
    UnitofMeasurement: STRING
    ReceivingDepartment: STRING
    Cost: DOLLARTYPE
  essential: OrdNum
  relationships:
    OrderedGood: received_item
  methods:
    Source( ):
    OrderedGood → received_item:
    DeliveredFrom → supplier
```

Clearly these two classes are very similar. One difficulty in integrating them by full structural correspondence is that they are members of (**memberof:**) different classes. Another difficulty is that purchase_order has a relationship to catalog_item, while receiving_report has a relationship to received_item. Later on we will show that the classes catalog_item and received_item can also be integrated by a common object type.

The integration of the classes purchase_orders and receiving_reports by a common object type is also possible, and in fact trivial. It is shown without further explanation in the Appendix. Therefore, the two classes purchase_order and receiving_report stand in full structural correspondence and can be represented by one common object type. This object type, FORM, represents the structural aspects of the two classes from the two databases.

```
objecttype FORM
  subtypeof: OPERATION
  memberof: FORMS
  attributes:
    OrderNumber: INTEGER
    Quantity: INTEGER
    Unit: STRING
    Cost: DOLLARTYPE
    FormDate: DATETYPE
    OrderingDepartment: STRING
  relationships:
    Item: ITEM
  methods:
    Supplier( ):
    Item → ITEM:
    OfferedBy → SUPPLIER
```

The following Table 1 shows the correspondence between the object types and the pairs of classes that are structurally integrated by these object types.

Table 1. Correspondence between two subschemas

PURCHASING_DEPARTMENT class	object type	INVENTORY_CONTROL class
purchase_order	FORM	receiving_report
purchase_orders	FORMS	receiving_reports
catalog_item	ITEM	received_item
supplier	SUPPLIER	supplier
operation	OPERATION	operation

The complete mappings for the object type FORM are given in Table 2. The properties of the object type FORM appear in the middle column. The result of the mapping M_1 (M_2) is shown in the left (right) column.

A short review of Table 2 shows that selectors for the class receiving_report are in general different from those of the object type FORM, while the class purchase_order shares most selectors with the object type FORM. The class receiving_report also has its attributes in a different order than the object type FORM. The fact that structural integration permits semantic differences is seen, for instance, in the lines dealing with FormDate. A delivery date (DeliveryDate) for an order is semantically quite different from the date when the order was filed (OrderDate). However, each of the two forms maintains only one of the those two dates, and they are structurally identical; therefore, they can be integrated into the single FormDate in the object type. Obviously, two classes that differ in the semantics of one of their attributes are themselves different in their semantics.

Another reason why purchase_order and receiving_report are semantically different is that they are different in their real life interpretation. FORM does not represent a class of objects that permits any real life operations common only to both classes but not common to classes representing other sheets of paper. In other words, in FORM we are not interested in operations such as "write on" or "tear up" which are common to all sheets of paper, but in operations that are genuinely specific to both purchase_order and receiving_report. We have not found such real life operations and claim, therefore, that FORM is not a correct semantic generalization of the two classes. Nevertheless, FORM summarizes interesting structural similarities.

Obviously, the probability of finding an application to which full structural correspondence can be applied is quite low. However, in practical applications we can expect some situations where integration by generalization is not feasible, but structural integration based on partial structural correspondence is possible. Structural correspondence was motivated by our work with actual telecommunication database schemas at Bellcore [24]. There, we demonstrated a realistic example that can be solved with attribute partial correspondence. In another practical application of partial structural correspondence [35], structural integration of a "student admission" subschema and a "candidate hiring" subschema of a university database is demonstrated. In both these applications, integration by generalization is not possible. In these examples, structural integration opens new possibilities for integration which go beyond the previously known integration techniques.

The reason that we need to discuss full structural correspondence in this paper is that the theory of partial structural correspondence is an extension of it, which cannot be easily understood without first comprehending the simpler case of full structural correspondence.

3.2.2. Structural integration of classes with partial correspondence. In many practical integration cases, one finds two classes that are different in only a few properties, and wants to integrate them in spite of their differences. Let us first formally define what *partial structural correspondence* between two classes a_1 and a_2 is.

Let Q_1 and Q_2 be the sets of properties of a_1 and a_2, respectively. Let $G_1 \subset Q_1$ and $G_2 \subset Q_2$ be the sets of properties for which there is a one-to-one correspondence, as defined for full structural

Table 2. Complete mappings for the object type FORM

purchase_order	FORM	receiving_report
(memberof:, purchase_orders)	(memberof:, FORMS)	(memberof:, receiving_reports)
(OrderNumber, INTEGER)	(OrderNumber, INTEGER)	(OrderNum, INTEGER)
(Quantity, INTEGER)	(Quantity, INTEGER)	(Quantity, INTEGER)
(Unit, STRING)	(Unit, STRING)	(UnitofMeasurement, STRING)
(Cost, DOLLARTYPE)	(Cost, DOLLARTYPE)	(Cost, DOLLARTYPE)
(OrderDate, DATETYPE)	(FormDate, DATETYPE)	(DeliveryDate, DATETYPE)
(OrderingDepartment, STRING)	(OrderingDepartment, STRING)	(ReceivingDepartment, STRING)
(Item, catalog_item)	(Item, ITEM)	(OrderedGood, received_item)
(Supplier, supplier)	(Supplier, SUPPLIER)	(Source, supplier)

correspondence. Then we say that the classes a_1 and a_2 have partial structural correspondence if the sets G_1 and G_2 are not empty.

Of course it is not useful to integrate a_1 and a_2 unless the cardinality of G_1 is close to the cardinality of Q_1 and Q_2. There are interesting special cases of partial structural correspondence, where the sets of not matching properties $Q_1 - G_1$ and $Q_2 - G_2$ include attributes only, relationships only, generic relations only or access path methods only. We refer to these special cases as *attribute partial correspondence relationship partial correspondence, generic relation partial correspondence* and *access path method partial correspondence*, respectively.

We concentrate in this paper on relationship partial correspondence. Attribute partial correspondence is similar and has been dealt with in a previous paper [24]. At the end of this section a short comment about the other types of partial correspondences will be made.

Suppose we are given two classes, a_1 and a_2, which exhibit relationship partial correspondence. Let $\mathscr{A}_1$ be the set of n_1 relationships of a_1, and let $\mathscr{A}_2$ be the set of n_2 relationships of a_2. Let $\mathscr{C}_1 \subset \mathscr{A}_1$ and $\mathscr{C}_2 \subset \mathscr{A}_2$ be the subsets of the relationships for which there exists a one-to-one correspondence (i.e. a full structural correspondence).

We now define an object type A common for both classes a_1 and a_2. We need to define the set of relationships $\mathscr{E}$ of $\mathscr{A}$ such that $\mathscr{E} = \mathscr{R} \cup \mathscr{S} \cup \mathscr{T}$, where $\mathscr{R}$ contains a relationship for each pair (q_1, q_2) of corresponding relationships $q_1 \in \mathscr{C}_1$, $q_2 \in \mathscr{C}_2$, $\mathscr{S} = \mathscr{A}_1 - \mathscr{C}_1$ and $\mathscr{T} = \mathscr{A}_2 - \mathscr{C}_2$.

Let $q_1 \in \mathscr{C}_1$ and $q_2 \in \mathscr{C}_2$ be two corresponding relationships in the partial structural correspondence between a_1 and a_2. By the definition of this correspondence, *typeof*(q_1) = *typeof*(q_2). We have to consider two cases:

1. selector(q_1) = selector(q_2). In that case, we define a relationship $r \in \mathscr{R}$ such that r = (selector(q_1), *typeof*(q_1)).
2. selector$(q_1) \neq$ selector(q_2). In this case, we define a relationship $r \in \mathscr{R}$ such that $r = (z,$ *typeof*$(q_1))$ where z = selector(q_1), or z = selector(q_2), or z is different from both.

We repeat this process for each such pair of relationships (q_1, q_2) satisfying the above conditions. For each relationship $q_1 \in (\mathscr{A}_1 - \mathscr{C}_1)$ we define for $\mathscr{S}$ a relationship s = (selector(q_1), *objecttype*(q_1)). For each relationship $q_2 \in (\mathscr{A}_2 - \mathscr{C}_2)$ we define for $\mathscr{T}$ a relationship t = (selector(q_2), *objecttype*(q_2)). To deal with the difference in the set of relationships of the object type A and its classes we specify two partial mappings. M_1 is a mapping from the set $\mathscr{E}$ of relationships of A onto the set $\mathscr{A}_1$ of relationships of a_1, and M_2 is a mapping from $\mathscr{E}$ onto the set $\mathscr{A}_2$ of relationships of a_2.

The cardinality of $\mathscr{E}$ is $|\mathscr{E}| = n_1 + n_2 - |\mathscr{C}_1|$. Thus, the mapping M_1 from $\mathscr{E}$ to $\mathscr{A}_1$ and the mapping M_2 from $\mathscr{E}$ to $\mathscr{A}_2$ are both mappings from a larger set to a smaller set. We define these two mappings from $\mathscr{E}$ by detailing the mappings for $\mathscr{R}$, $\mathscr{S}$ and $\mathscr{T}$.

For each relationship $r \in \mathscr{R}$, let (q_1, q_2) be the pair of corresponding relationships that exists according to our definition. Then

$$M_1((\text{selector}(r), \text{objecttype}(r))) = (\text{selector}(q_1), \text{class}(q_1))$$

$$M_2((\text{selector}(r), \text{objecttype}(r))) = (\text{selector}(q_2), \text{class}(q_2))$$

Note that by the definition of $\mathscr{R}$ both the classes class(q_1) and class(q_2) have the object type objecttype(r).

For each relationship $s \in \mathscr{S}$ and the corresponding relationship $q_1 \in \mathscr{A}_1$ the mappings are

$$M_1((\text{selector}(s), \text{objecttype}(s))) = (\text{selector}(q_1)\, \text{class}(q_1))$$

$$M_2((\text{selector}(s), \text{objecttype}(s))) = \text{NULL}$$

The class(q_1) has an objecttype(s) and selector(s) = selector(q_1).

For each relationship $t \in \mathscr{T}$ and the corresponding relationship $q_2 \in \mathscr{A}_2$ the mappings are,

$$M_1((\text{selector}(t), \text{objecttype}(t))) = \text{NULL}$$

$$M_2((\text{selector}(t), \text{objecttype}(t)) = (\text{selector}(q_2), \text{class}(q_2))$$

The class(q_2) has an objecttype(t) and selector(t) = selector(q_2).

We will now show an example of partial structural integration. The following object type ITEM integrates the class catalog_item from the purchasing department schema with the class received_item from the inventory control schema. For ease of reading, we first repeat these two classes here.

```
class catalog_item
  memberof: catalog_items
  attributes:
    Catalog Number: INTEGER
    Name: STRING
    Price: DOLLARTYPE
    Description: STRING
  essential: CatalogNumber
  relationships:
    OfferedBy: supplier
    (dependent) Orders: purchase_orders

class received_item
  attributes:
    CatalogNumber: INTEGER
    Price: DOLLARTYPE
    Name: STRING
    Damaged: BOOLEAN
  relationships:
    DeliveredFrom: supplier
    Reports: receiving_reports
```

The object type ITEM looks like:

```
objecttype ITEM
  memberof: ITEMS
  attributes:
    CatalogNumber: INTEGER
    Name: STRING
    Price: DOLLARTYPE
    Description: STRING
    Damaged: BOOLEAN
  relationships:
    OfferedBy: SUPPLIER
    Orders: FORMS
```

The complete mappings for the object type ITEM are given in Table 3.

Table 3 is characteristic of a partial structural correspondence. For instance, the "Damaged" attribute shown in the middle column exists for a received_item but not for a catalog_item, which can never be damaged. Therefore, the left column contains the entry NULL in the sixth row. Similarly, the *memberof* relation is not defined for the class received_item and the first row of the third column in Table 3 contains NULL.

Table 3. Complete mappings for the object type item

catalog_item	ITEM	received_item
(memberof:, catalog_items)	(memberof:, ITEMS)	NULL
(CatalogNumber, INTEGER)	(CatalogNumber, INTEGER)	(CatalogNumber, INTEGER)
(Name, STRING)	(Name, STRING)	(Name, STRING)
(Price, DOLLARTYPE)	(Price, DOLLARTYPE)	(Price, DOLLARTYPE)
(Description, STRING)	(Description, STRING)	NULL
NULL	(Damaged, BOOLEAN)	(Damaged, BOOLEAN)
(OfferedBy, supplier)	(OfferedBy, SUPPLIER)	(DeliveredFrom, supplier)
(Orders, purchase_orders)	(Orders, FORMS)	(Reports, receiving_reports)

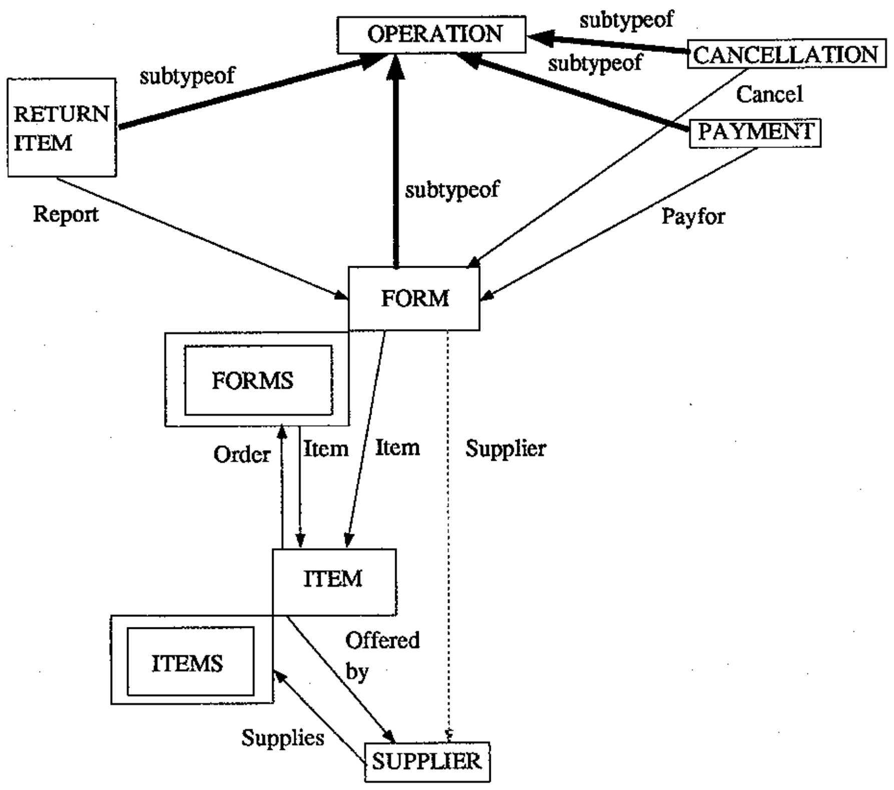

Fig. 3. Integration expressed by a schema of object types.

When we integrated the classes purchase_order and receiving_report, taking advantage of their full structural correspondence, it was pointed out that this integration would only be possible if the two classes catalog_item and received_item could also be integrated by one single object type. We have now accomplished this integration of catalog_item and received_item by defining the object type ITEM. The reader will notice that changing the order of presentation would not have helped in any way, because catalog_item and received_item both refer with relationships to purchase_order and receiving_report. In other words, the dependency is circular, and we can only integrate any one of the two pairs of classes if we can integrate the other pair also.

The case where one cannot integrate two classes *a* and *b* but *can* integrate two sets of classes **C** and **D**, such that $a \in \mathbf{C}$ and $b \in \mathbf{D}$ is quite common in structural integration. In the next section, we describe a procedure for identifying two such sets, i.e. two corresponding subschemas.

A complete description of the integrated database schema consists of the object types presented in this section, and the classes shown as code in Subsection 2.2 and graphically in Subsection 2.3. Because the object type description captures much of both schemas, it expresses the integration of the schemas for humans and for the system. Figure 3 shows the graphical representation of the object types.

The treatment of *generic relation partial correspondence* is very similar to that of relationship partial correspondence. The only difference is that for generic relations the correspondence is limited to identical relations, while for relationships different selectors are permitted. As an exception, the generic relations *categoryof* and *roleof* may correspond to each other.

Access path methods are chains of relationships and relations, possibly terminated by an attribute. Therefore, integration of access path methods follows the same principles as integration of relationships and does not warrant a repetitive description. The integration of operations will not be discussed in this paper and is the subject of future work.

4. FINDING SUBSCHEMAS WITH FULL STRUCTURAL CORRESPONDENCE

In order to ease the task of finding subschemas that permit structural integration, we now present a procedure for finding subschemas with full structural correspondence. For simplicity, this procedure ignores the use of methods, but a proper extension is possible. The data are the classes of two databases, D_A and D_B, which are to be structurally integrated.

In the process of structural integration, we are looking for pairs of classes with full structural correspondence. The first phase in this process is to look just for two such classes, one in each database. However, as was demonstrated in the previous section, most of the time such classes cannot be found due to their connections to other classes. Nevertheless, it may still be possible to find two subschemas with a one-to-one correspondence between their sets of classes.

The following procedure accepts human advice to avoid matching all permutations of two sets of properties which would increase its time complexity drastically. We also use the user's intuition in picking two classes for starting the matching process, rather than to run over all possibilities.

Once the procedure finds two corresponding subschemas it halts. This is the case when full structural correspondence between the two sets of classes is achieved and there exists no class in the subschemas with a relationship or relation to a class outside the subschemas. To continue searching for more corresponding subschemas, the procedure needs another pair of unmatched classes to start with.

The procedure uses two queues, Q_A and Q_B, and two arrays, R_A and R_B, to process the classes of the two desired corresponding subschemas, S_A and S_B. Initially, Q_A, Q_B, R_A and R_B are all empty.

Full_structural_correspondence (in D_A, D_B: schema; out R_A, R_B: schema)

1. Initialization: pick two object classes a_1 and b_1 which seem to be corresponding but are not identical in their properties. Insert a_1 into Q_A and b_1 into Q_B.
2. While the queues Q_A and Q_B are not empty do:

   Let $a_i(b_i)$ be the class at the front of $Q_A(Q_B)$. Transfer $a_i(b_i)$ from $Q_A(Q_B)$ to $R_A(R_B)$.
3. If the number of attributes in a_i and b_i is not equal, then exit(a_i, b_i). (No full correspondence is possible.) If the number of attributes of any given data type in a_i and b_i is not equal, then exit (a_i, b_i). (No full correspondence is possible.)
4. Consider the *categoryof* and *roleof*† relations of a_i and b_i. If their numbers are not equal, then exit(a_i, b_i). If a_i and b_i both have one such relation to object types a_j and b_j, respectively, then if a_j and b_j do not have identical object types or a_j and b_j do not appear as a pair in the arrays R_A and R_B, or they do not appear as a pair in the queues Q_A and Q_B, then insert a_j into Q_A and insert b_j into Q_B. (They are now candidates for comparison. If they appeared in R_A and R_B they were compared already. If they appeared in Q_A and Q_B then they are already waiting for processing.)
5. Multiple inheritance: if a_i has many *categoryof* and *roleof* relations to $a_{j_1}, a_{j_2}, \ldots, a_{j_m}$ and b_i has many *categoryof* and *roleof* relations to $b_{j_1}, b_{j_2}, \ldots, b_{j_m}$.

```
For $i = 1$ to $m$ do
  For $k = 1$ to $m$ do
    If $a_{j_i}$ and $b_{j_k}$ have a common object type
      or are stored as a pair in the array $R_A$ and $R_B$
      or are stored as a pair in the queues $Q_A$ and $Q_B$
    then delete $a_{j_i}$ and $b_{j_k}$ from the appropriate list $(a_{j_1}, a_{j_2}, \ldots, a_{j_m})$ or $(b_{j_1}, b_{j_2}, \ldots, b_{j_m})$.
```

6. Set m to the number of remaining object classes. Consider the two sets of remaining object classes $a_{j_1}, a_{j_2}, \ldots, a_{j_m}$ and $b_{j_1}, b_{j_2}, \ldots, b_{j_m}$. Display these two sets to the user so that he can suggest a one-to-one correspondence between these two sets and can rearrange the order of the b_{j_k} classes respectively, such that a_{j_k} corresponds to b_{j_k}, for $k = 1, \ldots, m$. Insert $a_{j_1}, \ldots, a_{j_m}$ into Q_A and $b_{j_1}, \ldots, b_{j_m}$ into Q_B.
7. Consider the *setof* relations of a_i and b_i. (There exists at most one.) The treatment is the same as for *categoryof* and *roleof* in step 4.
8. Consider the *memberof* relations of a_i and b_i. (There may be more than one.) The treatment is the same as for *categoryof* and *roleof* (in steps 4, 5 and 6).

†As before it is assumed that the *roleof* relation in question is annotated by a *subtypeof* relation.

9. Consider the relationships of a_i and b_i. The treatment is the same as for *categoryof* and *roleof* (in steps 4, 5 and 6).
   End of while loop (2).

The subschema S_A consists of the classes in R_A, and the subschema S_B consists of the classes in R_B, where the correspondence is given by the order in R_A and R_B. The previous procedure applies to the case of full structural correspondence. However, it is possible to modify it for cases of partial structural correspondence. The basic idea is as follows. If the given procedure fails, the user will be provided with the pair of classes causing the failure in question. If he decides that the two classes should actually correspond to the same object type, then the procedure can continue under the assumption of structural correspondence between these two classes. In such a case, the procedure can find two subschemas with partial structural correspondence.

In our example (Fig. 2), suppose the procedure starts with purchase_order in Q_A and receiving_report in Q_B. These two classes are transferred to R_A and R_B, respectively. The classes operation, purchase_orders and catalog_item are inserted into Q_A, and the classes operation, receiving_reports and received_item are inserted into Q_B. In the next steps the classes purchase_order and operation are moved into R_A, and the classes receiving_report and operation are moved into R_B. When processing catalog_item from Q_A and received_item from Q_B, we encounter a mismatch since catalog_item has a *memberof* relation to the class catalog_items, but received_item has no corresponding relation. Therefore, there is no full correspondence. However, if the user allows partial correspondence between catalog_item and received_item, the procedure can continue.

The class supplier is now inserted into both Q_A and Q_B. Processing this pair of classes leads to another mismatch because supplier has a relationship to catalog_items in the purchasing department schema, but there is no corresponding relationship in the inventory control schema. If we allow partial correspondence for the class supplier, we end up with two schemas with partial correspondence which are shown in Fig. 2 as two gray areas. Now one could try to re-execute the procedure with the initial pair of classes return_item and cancellation. One would find that either there is full structural correspondence, or there is attribute partial correspondence, depending on their lists of attributes which are not shown in this paper. The same applies to the pair return_item and payment.

5. CONCLUSIONS

The object-oriented Dual Model has been shown to be a good tool for structural integration. It separates the structure of an object-oriented database from its semantics. The semantic specification contains classes, and the structural specification contains object types, such that every class has one corresponding object type, but one object type may have several corresponding classes.

The classical method of integration is based on generalization, where, if two classes are to be integrated, a common superclass for them is created. However, it is not possible to find such a common superclass if the two classes to be integrated have different semantics. The Dual Model can integrate two classes even in this situation, because a single object type can correspond to several semantically different classes, as long as they are structurally similar.

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 even methods, thereby contributing to compact representations, software reusability and understandability. The structural schema also forms a "semantics free" representation that can be used as the basis for reusable optimization.

Structural integration is performed by defining a common object type for two corresponding classes and a mapping from the properties of the object type to the properties of each class. For a full structural correspondence, this mapping overcomes name and order differences and identifies the classes for the referenced object types in the relationships. The more complicated case of integration with partial correspondence has been formally defined, discussed and applied to a small but realistic example. Finally, a procedure for identifying subschemas of two databases which satisfy full structural correspondence was presented. Future work includes the formal definition of a corresponding procedure for partial structural correspondence and work on semantic integration. Currently, we are defining an OODB model for part relationships [36, 37]. Structural integration will have to be extended to deal with this part model.

Acknowledgements—We thank Mike Halper for proofreading a version of this paper and two anonymous reviewers for their thoughtful comments.

REFERENCES

[1] P. Ng, C. V. Ramamoorthy, L. Seifert and R. T. Yeh (Eds) *Proc. 1st Int. Conf. on Systems Integration*, Morristown, NJ. IEEE Computer Society Press (1990).

[2] P. Ng, C. V. Ramamoorthy, L. Seifert and R. T. Yeh (Eds) *Proc. 2nd Int. Conf. on Systems Integration*, Morristown, NJ. IEEE Computer Society Press (1992).

[3] M. Hammer and D. McLeod. On database management system architecture, Technical Report. MIT/LCS/TM-141, Massachusetts Institute of Technology, Cambridge, MA (1979).

[4] D. Heimbinger and D. McLeod. A federated architecture for information management. *ACM Trans. Office Inf. Syst.* **3**, 253–278 (1985).

[5] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.* **22**, 183–236 (1990).

[6] M. Rusinkiewicz, R. Elmasri, B. Czejdo, D. Georgakopoulous, G. Karabatis, A. Jamoussi, L. Loa and Y. Li. Omnibase: Design and implementation of a multidatabase system. In *Proc. A. Symp. in Parallel and Distributed Processing*, Dallas, TX, pp. 162–169 (1989).

[7] M. Templeton, D. Brill, A. Chen, S. Dao, E. Lund, R. McGregor and P. Ward. Mermaid: A front-end to distributed heterogeneous databases. In *Proc. of the IEEE*, pp. 695–708 (1987).

[8] R. Elmasri and S. Navathe. Object integration in logical database design. In *Proc. 1st Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 426–433 (1984).

[9] A. P. Sheth and J. A. Larson. A tool for integrating conceptual schemas and user views. In *Proc. 4th Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 176–183 (1988).

[10] M. L. Brodie and F. Manola. Database management: A survey. In *Readings in Artificial Intelligence and Databases* (Edited by J. Mylopoulos and M. L. Brodie). Morgan Kaufmann, San Mateo, CA (1989).

[11] W. Kim. Research direction for integrating heterogeneous databases. In *1989 Workshop on Heterogeneous Databases*, NSF, Northwestern University and IEEE-CS, pp. 1–5 (1989).

[12] M. Kaul, K. Drostern and E. Neuhold. Viewsystem: Integrating heterogeneous information bases by object oriented views. In *Proc. 6th Int. Conf. on Data Engineering*, Los Angeles, CA (1990).

[13] E. Bertino, M. Negri, G. Pelagatti and L. Sbattella. An object-oriented approach to the interconnection of heterogeneous databases. In *1989 Workshop on Heterogeneous Databases*, pp. 1–7, (1989).

[14] M. Schrefl and E. J. Neuhold. A knowledge-based approach to overcome structural differences in object-oriented database integration. In *Proc. of the IFIP Working Conf on The Role of Artificial Intelligence in Database and Information Systems*, Canton, China. North Holland, Amsterdam (1988).

[15] M. C. Shan. Unified access in a heterogeneous information environment. *IEEE Office Knowl. Engng* **2**, (1989).

[16] D. Fischer, W. Klas, L. Rostek, U. Schiel and V. Turau. VML—the vodak data modelling language. Technical Report GMD-IPSI (1989).

[17] E. J. Neuhold and M. Schrefl. Dynamic derivation of personalized views. In *Proc. 14th Int. Conf. on Very Large Databases*, Long Beach, CA, pp. 183–194 (1988).

[18] D. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. on Database Syst.* **6**, 140–173 (1981).

[19] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein and D. Beech. Third-generation data base system manifesto. Technical Report UCB/ERL M90/28, University of California, Berkeley (1990).

[20] E. J. Neuhold, J. Geller, Y. Perl and V. Turau. A theoretical underlying Dual Model for knowledge-based systems. In *Proc. of the 1st Int. Conf. on Systems Integration*, Morristown, NJ, pp. 96–103 (1990).

[21] E. Neuhold, Y. Perl, J. Geller and V. Turau. The Dual Model for object-oriented data bases. Technical Report CIS-91-30; New Jersey Institute of Technology. Submitted for publication.

[22] J. Geller, Y. Perl and E. J. Neuhold. Structure and semantics in OODB class specifications. *SIGMOD Rec.* **20**, 40–43 (1991).

[23] E. J. Neuhold, Y. Perl, J. Geller and V. Turau. Separating structural and semantic elements in object-oriented knowledge bases. In *Proc. of the Advanced Database System Symp.*, Kyoto, Japan, pp. 67–74 (1989).

[24] J. Geller, Y. Perl, P. Cannata, A. Sheth and E. Neuhold. A case study of structural integration. In *Proc. 1st Int. Conf. on Information and Knowledge Management*, Baltimore, MD. In press.

[25] J. Geller, Y. Perl and E. Neuhold. Structural schema integration in heterogeneous multi-database systems using the Dual Model. In *Proc. 1st Int. Workshop on Interoperability in Multidatabase Systems*, Los Alamitos, CA, pp. 200–203. IEEE Computer Society Press, Washington, DC (1991).

[26] V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore and O_2. *SIGMOD Rec.* **21**, 93–104 (1992).

[27] C. Lecluse and P. Richard. Modeling inheritance and genericity in object-oriented databases. Technical Report. LNCS #326, ICOT (1988).

[28] G. Copeland and D. Maier. Making Smalltalk a database system. *ACM SIGMOD* 316–324 (1984).

[29] 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* (Edited by R. Speth). North-Holland, Amsterdam (1988).

[30] D. Fishman, D. Beech, H. P. Cate and E. C. Chow. IRIS: An object-oriented DBMS. *ACM Trans. Office Inf. Syst.* **5**(1) (1987).

[31] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison–Wesley, Reading, MA (1983).

[32] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proc. ACM SIGMOD Conf. on Management of Data*, Washington, DC, pp. 340–355 (1986).

[33] M. Halper, J. Geller, Y. Perl and E. J. Neuhold. A graphical schema representation for object-oriented databases. In *Proc. Workshop on Interfaces in Database Systems (IDS-92)*, Glasgow. In press.

[34] W. Klas. A metaclass system for open object-oriented data models. PhD thesis, Technical University of Vienna, Vienna, Austria (1990).
[35] J. Geller, A. Mehta, Y. Perl, E. J. Neuhold and A. Sheth. Algorithms for structural schema integration. In *Proc. 2nd Int. Conf. on Systems Integration.* Morristown, NJ, pp. 604–614 (1992).
[36] M. Halper, J. Geller and Y. Perl. Part relationships for object-oriented databases. In *Proc. the 11th Int. Conf. on the Entity Relationship Approach*, Karlsruhe. In press.
[37] M. Halper, J. Geller and Y. Perl. An OODB part relationship model. In *Proc. 1st Int. Conf. on Information and Knowledge Management.* In press.

APPENDIX

Although the parts of the integrated subschema have occurred in the body of the paper, they have not occurred in the correct order and in a comprehensive form. Therefore, in this appendix, we present now the object types and classes of the integrated subschema. A quick comparison with Fig. 2 will show that all classes not relevant to the integration process have been omitted. We also show the object type FORMS which was omitted in the body of the paper and which integrates the classes purchase_orders and receiving_reports.

```
objecttype FORM
  subtypeof: OPERATION
  memberof: FORMS
  attributes:
    OrderNumber: INTEGER
    Quantity: INTEGER
    Unit: STRING
    Cost: DOLLARTYPE
    FormDate: DATETYPE
    OrderingDepartment: STRING
  relationships:
    Item: ITEM
  methods:
    Supplier( ):
    Item → ITEM:
    OfferedBy → SUPPLIER

class purchase_order
  objecttype: FORM
  categoryof: operation
  memberof: purchase_orders
  attributes:
    OrderDate: DATETYPE
  essential: OrderNumber
  relationships:
    Item: catalog_item
  methods:
    Supplier( ):
    Item → catalog_item:
    OfferedBy → supplier

class receiving_report
  objecttype: FORM
  categoryof: operation
  memberof: receiving_reports
  attributes:
    OrdNum: INTEGER
    DeliveryDate: DATETYPE
    UnitofMeasurement: STRING
  essential: OrdNum
  relationships:
    OrderedGood: received_item
  methods:
    Source( ):
    OrderedGood → received_item:
    DeliveredFrom → supplier

objecttype ITEM
  memberof: ITEMS
  attributes:
    CatalogNumber: INTEGER
    Name: STRING
    Price: DOLLARTYPE
    Description: STRING
    Damaged: BOOLEAN
  relationships:
    OfferedBy: SUPPLIER
    Order: FORM

class catalog_item
  objecttype ITEM
  memberof: catalog_items
  essential: CatalogNumber
  relationships:
    (dependent) Orders: purchase_orders

class received_item
  objecttype: ITEM
  relationships:
    DeliveredFrom: supplier
    Report: receiving_report

objecttype FORMS
  setof: FORM
  attributes:
    NumberofForms: INTEGER
    GroupPurpose: STRING
  relationships:
    Item: ITEM

class purchase_orders
  objecttype: FORMS
  setof: purchase_order
  attributes:
    NumberofOrders: INTEGER
  relationships:
    Item: catalog_item

class receiving_reports
  objecttype: FORMS
  setof: receiving_report
  attributes:
    Quantity: INTEGER
  relationships:
    Item: received_item
```