iCSi '90

The First International Conference on Systems Integration

April 23-26, 1990 Headquarters Plaza Hotel, Morristown, New Jersey, USA

Edited by

Peter A. Ng
New Jersey Institute of Technology

C.V. Ramamoorthy
University of California, Berkeley

Laurence C. Seifert
AT&T Bell Laboratories, Inc.

Raymond T. Yeh
Syscorp International, Inc.

Sponsored by
Institute for Integrated Systems Research
Department of Computer & Information Science
New Jersey Institute of Technology (NJIT)

In cooperation with
IEEE Computer Society
ACM (Association for Computing Machinery)
AT&T
Bell Communications Research (Bellcore)
Gesellschaft Fuer Mathematik und Datenverarbeitung (GMD)

This conference focuses on the problems, issues and solutions of integrated system design, implementation and performance. Integration technologies will be emphasized with a focus on computer-aided software engineering, collaborative and distributed systems, and computer integrated manufacturing systems. The conference will provide an international and interdisciplinary forum in which researchers and practitioners can share novel research, engineering development and strategic management experiences.

At the core of the program are 85 selected papers which present recent advances on many aspects of systems integration technologies. The program will be highlighted by three plenary session speakers, a banquet speaker, and four panels addressing emerging issues of widespread interest and bringing a variety of experiences and opinions to the conference. In addition, there is a one-day program of five tutorials which presents a variety of important topics.

IEEE Computer Society Press
Los Alamitos, CA

Washington • Brussels • Tokyo

A Theoretical Underlying Dual Model for Knowledge-Based Systems

Erich Neuhold+ James Geller* Yehoshua Perl*
Volker Turau+

+GMD-IPSI Integrated Publication and Information Systems Institute
Dolivostr. 15, D-6100 Darmstadt, FRG
*Institute for Integrated Systems, Department of Computer Science
New Jersey Institute of Technology, Newark, New Jersey 07102

Abstract

Object-oriented knowledge-based approaches have proved to be very powerful vehicles when developing, respectively integrating, complex systems. Their abstraction and inheritance capabilities encourage proper modularisation as well as support the encapsulation and incorporation of previously existing system components that have not been developed following the object oriented paradigm.

Unfortunately, many different definitions of the term knowledge-based exist, and in most of them some terms are only "defined" by example or by implementation. In this paper we show how a theoretical underlying model for object-oriented knowledge-based systems can enhance the precise understanding of its modeling power and consequently simplify the sharing of system components by the developers of the systems. This is one of the major problems encountered in todays large knowledge-based systems that has led to loss of modularity and extendibility, the prime motivators to move to object-oriented approaches. Since a formal model for all approaches is impossible, we concentrate on the typed, extensible object-oriented data-model developed at GMD-IPSI.

Theoretical underlying models for other object-oriented knowledge base systems can be obtained in a similar way. The integration of different knowledge-based systems will be much easier when the theoretical underlying model for each one of them is well defined.

In this paper we concentrate on defining and cleanly separating the structural and semantic parts of object-oriented definitions. Here *structural* refers to the organisation of the data and the operations on the data whereas *semantic* refers to the connections of these concepts to the real world.

1 Introduction

The large complex systems that are currently under development usually suffer from the incompatibility of the many interfaces that have to be handled. Unfortunately we are usually not free to redesign those interfaces if we want to import preexisting system functions - a must in most systems under development today. Usually we have to deal with many different kinds of information, many different types of representations of this information and many different operations that may be used to manipulate the data.

The object oriented paradigm which combines more traditional approaches like abstract data types, inheritance structures and semantic data modeling principles has been used as a powerful mechanism to rapidly develop and integrate interfaces of subsystem components. The class construct of the object oriented paradigm allows for easy categorisation of objects via structural and semantic similarity. Inheritance provides for reusability and information hiding for modularization. Usually, however, these concepts are not connected to features like persistency, sharability, and consistency, properties that in traditional systems are handled by data base management systems.

We have described an object oriented data model [NPGT89a, NPGT89b, FKRST89] - and an accompanying knowledge base management system is currently being built - that will be the basis for

a. the integration of the data and data operations required for the multitude of applications appearing in complex systems [NS88];

b. interconnecting the object-oriented paradigms of the programming and artificial intelligence world with object oriented data models [NK89];

c. integrating preexisting applications and data bases with new knowledge-based components into a single object-oriented knowledge-based system [SN88a, SN88b].

In this paper we want to provide a formal basis for an object oriented data model, since we feel that only on a precisely defined basis can we build a reliable technology for integrating the multitude of data manipulation needs of the applications found in the complex systems of today and the future.

In order to store and retrieve the information for an application from a certain enterprise's database, a suitable representation of the application's environment is required. Such a representation is called a conceptual model. It is an abstract representation that contains the properties of the environment relevant to the application. In the past years numerous models with the aim of representing semantic structures beyond the capabilities of the first-normal-form relational model were presented [PM88]. The representation mechanisms are all based on a collection of abstraction techniques. The most significant technique is to collect the objects of the domain of interest into classes. The class concept has been introduced in Simula [DN66] and reimplemented by Smalltalk [GR83].

A class can be regarded as a container for objects which are similar in their structure and their semantics in the enterprise. Furthermore, the objects have a similar behavior from the user's point of view. Therefore, a class can be regarded as a description of the structure and behavior of objects.

There still exists confusion regarding the meaning of some terms used in different object oriented systems. Different systems use different terminology and lack exact definitions for the terms used. Some terms are vaguely described using other undefined terms and the description is then supported only by examples which demonstrate the meaning of the terms. This situation causes difficulties both in understanding and implementing these systems. Furthermore these issues make the communication between people using different systems more difficult. To overcome these difficulties we present mathematical definitions for the terms used in our object oriented database system [FKRST89]. Such definitions exist already for some basic terms [DKT88] but still are needed for the more complicated terms. In other words we are trying to establish a theoretical underlying model for object oriented knowledge-based systems.

In such a theoretical underlying model there should be a mathematically defined structure corresponding to each term used in the system. This will help in overcoming the above mentioned confusion encountered in the implementation and the integration of such systems and in the human communication about different systems using similar but not identical terminology. With formal definitions, for example, the identity of different terms could be judged according to their corresponding mathematical structures. For example, what are the mathematical structures corresponding to method chains and transformer chains (defined later in the paper), or statements as "class A is a subclass of class B", "class A is a category of class B", "class A is dependent on class B" or "class A is a member of class B"?

But, let us explore now another source of confusion in object oriented knowledge and database systems. Following [W87], the characteristic concepts of object-oriented languages are the notions of objects, classes of objects, and inheritance of properties between objects. The description of an object class contains both structural parts and parts that describe the semantics the classes have in the application. In many systems, e.g. [CM84, KNS88, F87, GR83, SR86], the structural and semantic parts are mingled together. This causes some difficulties when it is necessary to distinguish whether a part is structural or contains semantic information.

An example for such a difficulty is the question whether an operation on the elements of a class (a method) is a structural or semantic element of a class. Similarly, do the following statements contain structural or semantic information: "class A is dependent on class B", "class A is role of class B", "class A is a set of class B"?

In order to approach these problems we will first present our view of the basic elements of object oriented data and knowledge base system. To provide reusability, classes are organised in a class hierarchy. Sublcasses inherit the properties of their super classes. In many systems as O2 [LR88] and Vbase [AM87] the subclass hierarchy is used for two purposes at the same time:

a) to factorise common structure and behavior of classes and

b) to express additional semantic relationships between classes.

This leads to a situation that two classes modeling semantically related objects could only be dealt with, if the objects in question are structurally related as well. The use of a single hierarchy for two conceptually distinct connections among specifications resulted in inadequate conceptual models. Therefore, it will be advantageous to separate those two parts of the specification.

In [NPGT89a, NPGT89b] we recently introduced the so called Dual Model where we separate in the definition of the object class the structural parts from those parts describing the semantics of the class in the real world. In order to express that all instances of a class have a common structure and behavior we consider them to be of the same abstract data type. This type is called the *object type* of that class.

Hence, we associate with each object class an object type. Regardless of their corresponding object type, object classes can be related according to their semantic connections in the application, which can be formulated independently of the object types.

For example it may be desirable to represent the same object in different contexts of the application, or to deal with an object in different levels of detail. All the semantic constraints are now expressible in the model, regardless of the structural relations between the relevant classes.

Some object-oriented programming languages use the concept of *abstract classes* [GR83,MSOP86], which are classes which cannot be instantiated. Their only purpose is to define useful attributes, relationships, and methods that can be inherited by concrete subclasses. We achieve the same result by defining object types which include these attributes, relationships, and methods and then utilise type inheritance. In this way the anomalous concept of abstract classes is avoided.

Thus, our specification allows two hierarchies, on the one side we have a structural hierarchy (an acyclic directed graph) of object types, and on the other a network of classes. In making this distinction we hope to achieve a better abstraction mechanism, giving a more accurate representation of the application.

In this paper we want to combine the precise mathematical description of database concepts with the Dual Model of structure and semantics as presented in [NPGT89a, NPGT89b]. Since both these descriptive methodologies are intended to increase the clarity of our database model, we expect the combination to resolve the confusion existing with regards to the precise meaning of certain technical terms and in identifying whether a statement is discussing structural or semantic information.

The separation of structural and semantic elements in the Dual Model leads initially to more difficulties in finding mathematically precise definitions for technical terms. However, after overcoming these difficulties, a clear and precise model emerges.

We will use this paper also to discuss our reasoning in determining whether an element in the description of an object class is structural or contains semantic information. This reasoning was partially omitted in [NPGT89a, NPGT89b] due to space limitations, but it is important to properly understand the details of the Dual Model.

In Section 2 we review the idea of "object type" as previously presented in [NPGT89a, NPGT89b]. The basic properties of object classes are considered in Section 3. The difference between these two descriptions is that in object types we consider only the structural elements of the class description, while the semantic elements are considered in the description of the object class.

In Section 4 we consider structural connections between object types showing them to be implicit relations. In Section 5 we consider semantic connections between object classes, showing them also to be implicit relations. The distinctions between structural and semantic implicit relations is discussed. Finally, an example in Section 6 demonstrates the different elements of the Dual Model.

2 Basic Properties of Object Types

In order to structure the set of objects in the domain of our interest we collect objects into classes. An object class can be regarded as a container for objects which are similar in their structure and their semantics in the real world. They have similar behavior from the user's point of view.

An object is said to belong to the class or be an instance of the class. The actual set of objects which belong to a given class is called the extension of the class. The set of all possible objects of this class (possibly an infinite set) is called the domain of the class.

Data types are a major organising principle for programming languages [DT88]. They describe the common properties of objects. A type describes a (possibly infinite) set of values and a set of operations applicable to these values. Moreover, a variable of that type is constrained to assume values of that type.

Since all objects of a class have a common structure, they will have a common representation. Therefore, they will be considered as instances of the same object type. This type is called the object type of that class. However, the objects of two different classes may be of the same object type, even though the two classes model objects having different semantics in the real world. Thus the description of the object type of a class does not reflect the semantics which the objects carry in the real world, it is purely a structural description of the representation for the instances and can be separated from the class description to form an object type specification.

The separation of the object type description from the description of the object class, i. e. what we call the "Dual model", will help to get a clear distinction between the structural and semantic parts in the definition of an object class. The semantics of the objects is described in the definition of the object classes by relating them to the object types and by determining their (semantic) connections with other classes. Hence, the object types only serve as a platform for specification of the semantics, but do not contain the semantics themselves.

As usual we allow three different kinds of properties to be defined for a class:

1. Attributes, with values of some type,

2. Relationships, with references to another object class,

3. Methods, to be used on the instances of the object class.

We will now distinguish which of these three kinds of properties are structural and which contain semantic information. Attribute definitions which describe the type of a data element are clearly structural. Relationships contain semantic information as they refer to another class which has semantic information in the context of the application.

The determination of the status of methods is more involved. Transformer chains are structural, while method chains (both defined later in this Section) refer to object classes which contain semantic information and thus also seem to contain semantic information. As defined later, a method can be either of the above two, or a composition of two such chains. Thus it is not clear whether a method is structural or is a semantic part in the definition of a class. (We suggest that the reader review this discussion after reading section 2.3 about methods).

When we define an object type we should include only structural elements of the definition of a class in it. Thus we have to resolve the difficulty regarding methods. Furthermore we need to find a structural way to represent relationships, since there is no sense in defining object types if they do not contain relationships. (Without relationships we would only have the definition of a (complex) abstract data type as known from programming languages.)

Fortunately, these difficulties are resolved easily once we permit the object type to contain the structural elements of the object class. In an object type a relationship is defined as referring not to an object class, but to an object type and thus is structural and as such can be included in the object type. Furthermore, a method chain in an object type refers only to object types and thus is also structural, as is a transformer chain. For both, relationships and methods, the reference to the actual object classes is contained in a class definition which corresponds to the object type definition and has meaning in the context of the application at hand. (See the examples for methods in Section 6).

This means that structural relationships and methods are defined in object types and refer to object types, while the corresponding semantic relationships and methods are described in the object class, substituting each object type in the relationship or method by the corresponding object class. We call our model the "Dual Model" due to the separation of the class description into two layers, the structural layer and the semantic layer.

As a matter of fact, the separation of the definition of methods into two layers, the structural and the semantic layer resolves the difficulty of determining whether a method is structural or semantic in nature, which we have raised earlier. We see this resolution as one of the main advantages of our separation.

Thus we are ready for the formal description of an object type. An object type is determined by a list of properties. There are three different kinds of properties and they correspond to the already introduced class properties:

1. Attributes, with values of some type,
2. Relationships, with references to other object types,
3. Methods, to be used on the instances of the object type.

2.1 Attributes

An attribute property is represented by a name and a data type. The name is a selector for this property. The type specifies a domain of values from which a value for this attribute can be chosen. Note that a type can be defined as a composite type of sets, enumeration types, tuples, and disjoint unions, as described in [NPGT89a, NPGT89b]. The mathematical structure underlying an attribute is a variable. This conforms with the behavior of an attribute to refer to one value of a data type at one time.

2.2 Relationships

A relationship property is represented by a name and an object type. The name is again a selector for the property. The object type referenced must be defined elsewhere. In a relationship we allow not only a single object type but also a composite structure of object types using the same set, tuple, and disjoint union constructors that we used in defining composite data types [NPGT89a, NPGT89b]. In this way we can refer to a set or tuple of object types in a relationship.

Usually in object oriented programming models the underlying mathematical structure for a relationship from object class A to object class B is a mathematical relation between the extensions (i.e. the sets of instances) of the classes A and B. Now, when we need to define the underlying mathematical structure for the structural relationship of an object type we encounter some difficulty, since an object type, by definition, does not have an extension. However, an object type has a domain which is a set of instances, namely all *possible* instances, and we use the domain to provide the underlying mathematical structure as follows.

Let R be a relationship from an object type A to an object type B. Let $D(A)$ and $D(B)$ be the domains of the object types A and B respectively. The relationship R is a relation from the set $D(A)$ to the set $D(B)$, referring here to the mathematical notation of relation as a subset of the Cartesian product
$D(A) \times D(B) = \{ (a , b) / a \in D(A), b \in D(B)\}$.
Hence, the relationship is a set of ordered pairs of objects.

In this definition we used the fact that the domain of an object type is a structural entity, while the extension of the corresponding object class represents semantic information. The definition of the domain of an object type is given in Section 2.4.

2.3 Methods

In the following we use the Smalltalk [GR83] terminology for methods but have redefined some of its meaning. A method is a program segment with one required parameter of some object type, and any number of optional parameters. We will assume that every method also returns a value of an object type or data type. The method name together with these optional parameters is called the "message" which is sent to the object identified by the required parameter.

If a program segment is needed that takes values of a data type as arguments then it must be defined as an operation of this data type rather than a method, and it will also return a value of a data type.

The signature of a method or of an operation defines the types of its input parameters and the type of the return value. The definition of an object type contains a signature for every method that is defined for that object type as its required parameter.

In the following we will formalise methods that do not perform any side effects in persistent memory, i.e. methods that perform only local computations and that influence their environment only by their return value. We will give a recursive definition of such a method.

A *computational method* is a program segment with one required parameter of some object type that makes use of the functionality of the underlying programming language (e.g. C++) but does not modify any stored values outside of its own local memory and returns a value of an object type.

A *primitive method* is either a relationship or a computational method.

A *method chain* is either a primitive method or a primitive method composed with a method chain.

A *transformer* is a program segment that takes as a required argument a value of an object type and returns a value of a data type. Other than that it behaves like a computational method.

A *primitive transformer* is either a transformer or an attribute.

An *operation chain* is either an operation or an operation composed with an *operation chain.* (Operations are defined for "ordinary" abstract data types).

A *transformer chain* is either a primitive transformer or a primitive transformer composed with an operation chain.

With all these terms in place we can now define a method formally. A *method* is either a method chain or a transformer chain or a composition of the two, namely a method chain composed with a transformer chain.

According to our definition relationships and attributes are just special cases of methods. Nevertheless, they are conceptually important special cases which warrant our three way distinction between attributes, relationships, and methods.

We will refer to the set of all possible argument values of one parameter as the *domain* of this parameter. The *domain of a method* is the cross product of the domains of all its parameters. We define the *range of a method* as the set of all results that this method can generate given all possible values of its domain as inputs.

With the above assumptions one can view a method with no side effects as a relation. This relation will be from the domain of the method to its range.

$$\text{R: Domain} \rightarrow \text{Range}$$

In case that the domain is a cross product this relation takes the following form. Let Dom_i be the domain of the i-th argument of the method.

$$\text{R: Domain(Object type)} \times Dom_1 \times Dom_2 \ldots \times Dom_n \rightarrow \text{Range}$$

This basic idea can be carried over easily to all elements that have been used in the formal definition of a method. For an operation we get

$$\text{R: Data type} \times Dom_1 \times Dom_2 \ldots \times Dom_n \rightarrow \text{Data type.}$$

For a transformer we get

$$R: \text{Domain(Object type)} \times Dom_1 \times Dom_2 \ldots\ldots \times Dom_n \rightarrow \text{Data type}$$

and for a computational method

$$R: \text{Domain(Object type)} \times Dom_1 \times Dom_2 \ldots\ldots \times Dom_n \rightarrow \text{Range} = \text{Domain(Object type)}.$$

Our definition of a method permits the chaining of the above defined entities in the following pairs. (1) For method chains: computational method - computational method, relationship - relationship, relationship - computational method, computational method - relationship. (2) For transformer chains: operation - operation, attribute - operation, transformer - operation. (3) Chaining method chains with transformer chains creates the following additional possible pairs: computational method - operation, computational method - transformer, computational method - attribute, relationship - operation, relationship - transformer, and relationship - attribute.

It is obvious that with our restrictions all the mentioned compositional pairs can be represented as the compositions of relations.
E. g. If R_1: A → B is a relation from A to B and R_2: B → C is a relation from B to C then we can define the composite relation R_3: A →C as

$$R_3 = R_1 \circ R_2$$

such that "o" defines relation composition in the usual sense:

$$R_1 o R_2 = \{(x,\ y)|\ xR_1z\ \&\ zR_2y\}$$

Clearly our recursive definition covers composition chains of any length, e.g. if a method R is defined by chaining relationships R_1, R_2, R_3 to an attribute A_1, then R = $(((R_1 \circ R_2) \circ R_3) \circ A_1)$. Therefore, a method chain, an operation chain, a transformer chain, and thus a method can each be represented as a composition of relations. For examples of object types see Section 6.

2.4 The Domain of an Object Type

We need to define the domain of an object type. This domain should reflect the attributes, relationships, and methods of the object type. To facilitate the definition we define a *selector Cartesian product* which is a variation of the regular Cartesian product.

A Cartesian product $A_1 \times A_2 \times \ldots A_n$ is the set of all n tuples $(a_1, a_2, \ldots, a_i, \ldots, a_n)$ such that $a_i \in A_i$ for $1 \leq i \leq n$. That is, an element of the Cartesian product is a mapping M from the tuple $(1, 2, \ldots, n)$ to an n-tuple such that $M(i) = a_i$, where $a_i \in A_i$ for $1 \leq i \leq n$.

In an object type the properties are identified by their selector, rather than by a serial number. Thus we define a selector Cartesian product $A_{PROPERTY1} \times A_{PROPERTY2} \times \ldots \times A_{PROPERTYn}$ as a mapping M from the tuple of the properties of the object type to the Cartesian product of n sets as follows:

$$M : (PROPERTY1, PROPERTY2, \ldots, PROPERTYn) \rightarrow A_{PROPERTY1} \times A_{PROPERTY2} \times \ldots \times A_{PROPERTYn}.$$

An element of the selector Cartesian product is a mapping

$$(PROPERTY1, PROPERTY2, \ldots, PROPERTYn) \rightarrow (a_{PROPERTY1}, a_{PROPERTY2}, \ldots, a_{PROPERTYn}),$$

where

$$a_{PROPERTYi} \in A_{PROPERTYi} \text{ for } 1 \leq i \leq n.$$

We still have to define the $A_{PROPERTYi}$. If $PROPERTYi$ is an attribute then $A_{PROPERTYi}$ is the type defined for this attribute in the definition of the object type. If $PROPERTYi$ is a relationship to another objecttype OT, then we define $A_{PROPERTYi}$ to be the set of all possible classes whose object type is OT. If $PROPERTYi$ is a method M then we define $A_{PROPERTYi}$ to be the method M itself. Thus the domain of the object type ORDER in Section 6 is the mapping $(Orderdate, Quantity, Orderingcustomer, Customername) \rightarrow (DATE \times\ INTEGER \times\ \{oc/oc\ \textit{is a class with the object type}\ CUSTOMER\} \times \textit{the method Customername})$

To justify our choice of $A_{PROPERTYi}$ where PROPERTYi is a relationship R from an object type A to an object type B let us consider an alternative. The natural alternative is that $A_{PROPERTYi} = D(B)$, where $D(B)$ is the domain of B. But B is an object type which may have a relationship S to the object type A. In this case we obtain a cycle in the definition and the domain of the object type A is not well defined. Thus we have chosen the above alternative.

3 Basic Properties of Object Classes

So far we have concentrated on the specification of the object types. Classes were only used as an explanatory tool to derive the definitions of the properties of object types. Now we will deal with details of classes. An object class specifies the semantics of an object in the real world by identifying relationships to other object classes, the constraints on the relationships and the way the methods specified with the object type have to be applied to the objects in the environment of the existing object classes. In the definition of the object type the relationships only relate different object types. In the semantic description we must say which object classes of this object type we actually want to relate. This is necessary if more than one object class has the same object type, otherwise this information is implicitly contained in the object type definition. The same must be done for the object types in the signatures of the methods.

We now need to present the underlying mathematical structures for the properties of object classes. Actually, these mathematical definitions are common knowledge in object oriented programming and are briefly repeated here for reasons of completeness.

Suppose an object class A is introduced as being of object type A' for which a (structural) relationship R' to object type B' is defined. Let B be the object class corresponding to the object type B'. Then, by stating that object class A is of object type A' it is implied that it has a semantic relationship R to the object class B. The mathematical underlying structure of this relationship is a relation from the extension $E(A)$ of A to the extension $E(B)$ of B, referring here to the mathematical notation of "relation" as a subset of the cartesian product of the two extensions.

$$E(A) \times E(B) = \{(a, b)/a \in E(A), b \in E(B)\}$$

Hence the relationship is a set of ordered pairs of instances of A and B.

As can be seen, this definition is analog to the definition of the structural relationship just replacing the structural set of the domain by the semantic set of the extension. This definition is, as stated before, common knowledge. The contribution of our research is the structural definition, but, because an object class is defined as being of a given object type, the class definition is presented second in our description.

The same applies with regards to the definition of the semantic methods. We just need to replace the structural relationships in the definition of the structural methods by the corresponding semantic relationships to obtain the mathematical underlying structure for a semantic method. We omit repeating the details of this definition.

Now we can define additional 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 all different from nil.

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 the former). Then the deletion of this object has the consequence, that the dependent objects are also deleted.

4 Connections between Object Types

In addition to the basic properties of object types and object classes their descriptions may contain statements about their connections to other object types or object classes. Examples of such statements are: "class A is a subclass of class B", "object type A is a subtype of object type B", "class A is a category of class B", "class A is a role of class B", "class A is a member of class B", "class A is a set of class B". These statements are the ones which we referred to in our introduction as being not well defined for some systems. For each of these statements we need to identify whether the connection described is structural or contains semantic information according to the separation expressed by our Dual Model. Furthermore we need to define the underlying mathematical structure corresponding to each such statement.

Note that we already defined in Section 3 that the statements "Class A is dependent on class B", "attribute X is essential", and "relationship Y is essential" are semantic constraints on properties. Thus we do not need to treat these kinds of statements here. In the following we continue to denote by A' the object type of the class A.

Structural information is contained in the following statements.

1. Object type A' is a subtype of object type B'.

2. Object type A' is a member of object type B'.

3. Object type A' is a set of object type B'.

Because these statements are structural they will be contained in the description of an object type. By introducing the notion of object type we do not need any "subclass statements" since the information communicated by such a statement is structural and is now given by the subtype statement in the description of the object type.

We shall show that each one of these statements can be described as a relation. As we mentioned before, the relationship is also a relation. However, we shall show that there is a difference between the relations which describe relationships and these relations.

There are two ways in which one can represent the pairs belonging to a relation R from a set B to a set A. One way is to list all the pairs explicitly. The second way is by specifying a rule which determines all the pairs of a relation without listing them. That is, for each pair (b, a) $b \in B$, $a \in A$ it is possible to determine if it belongs to R. Examples of the second way are the relations DIVIDE and GREATER from INTEGER to INTEGER. We call such a representation of a relation an implicit representation, since the pairs are not listed explicitly. Actually every infinite relation must have an implicit representation since it is impossible to list explicitly an infinite sequence of pairs. Note that there is no difference between relations which are represented by one way or another. The only difference is in their representation. For convenience we refer to a relation represented implicitly as an *implicit relation*.

We shall show that each of the above statements expresses an implicit relation.

4.1 The Subtype Relation

We now introduce subtypes of object types [KNS88, FKRST88]. If several object types share common characteristics, then one can factor out the commonality to produce a more general object type. One could then include the definition of this general object type into the definition of the other object types. The general object type is called supertype of the given types, which are called its subtypes. A subtype describes an object type that is more specific than the supertype. Whether the subtype is regarded as a specialization of the supertype or the supertype is regarded as a generalization of the subtype depends on the point of view and is not relevant to our discussion. This mechanism of organizing object type descriptions on several levels can generate a hierarchy of object types. The supertype relation is transitive. To distinguish the case where the relation is given rather than composed we use the term *immediate supertype*. The primary motivation for the use of subtypes is that it provides tools for both structuring and reusing specifications. To find the commonality between object types is a crucial task in developing an application.

Structural inheritance is the mechanism which uses the commonality specified in the subtype hierarchy of object types. The subtype inherits all the properties (i.e. attributes, relationships and methods) which are defined for its supertypes. In particular the methods defined with the supertype are also applicable to elements of the domain of the subtype. Inheritance is achieved by using a special kind of *coercion*. A coercion is a projection, that is a mapping from a tuple into a subset of the elements of this tuple [DKT88]. Note that this is a mathematical definition of coercion rather than the programming language meaning of coercion. As we saw in the definition of the domain of an object type (Section 2.4) the elements of the tuple are identified by selector names rather than indices. Thus we need to define a selector coercion similar to the selector cartesian product as a mapping

$$(PROPERTY_1, PROPERTY_2, \ldots, PROPERTY_n) \rightarrow$$
$$(PROPERTY i_1, PROPERTY i_2, \ldots, PROPERTY i_k)$$

where $\{i_1, i_2, \ldots, i_k\}$ is a subset of $\{1, 2, \ldots, n\}$. In our case the selector coercion is a mapping from the domain of an object type $(A_{PROPERTY1}, A_{PROPERTY2}, \ldots, A_{PROPERTYn})$ to the domain of its supertype $(A_{PROPERTYi1}, A_{PROPERTYi2}, \ldots, A_{PROPERTYik})$. Now the methods actually available for an element of the domain of a particular object type are not only those methods defined in the element's object type but also those of all its supertypes in the object type hierarchy.

Thus the subtype connection is a set of pairs of elements, the first is an instance of the domain of the object type A' and the second is an instance of the supertype object type B'. Thus the subtype connection is actually a relation from the domain of A' to the domain of B'. However, the pairs of the relation do not need to be listed as they are described implicitly by the coercion. Hence subtype is an implicit relation. Note that as for the structural relationships and methods the relation is defined between the domains of the object types and not the extensions of the corresponding classes.

Further discussions of the structural subtype hierarchy and the structural inheritance can be found in [NPGT89a, NPGT89b].

4.2 The Set Of and Member Of Relations

Consider two object types A' and B' such that the connection between them is described by the statement "A' is a set of B' ". That is, the instance of the domain of the object type A' is actually a set of instances of the domain of the object type B'. For example, when B' describes a member of an organisation and A' describes a committee of this organisation which contains a set of members.

This is the description of the connection from the point of view of A'. From the point of view of B' the same connection is described by the statement "B' is a member of A' ". These two statements contain structural information since a set is a mathematical structure without semantic information.

The mathematical structure which describes the connection "set of" between the object types A' and B' is a relation from the domain of A' to the domain of B'. Note that the domain of A' is the power set of the domain of B'. The relation contains a set of pairs (a, b) such that a is an instance of the domain of A', or in other words, a is a finite subset of the domain of B', and b is an element of the set a, using the mathematical notion of set and element. This is also an implicit relation since the pairs of the relation are not listed, they are described implicitly by the mathematical concept of membership of an element in a set.

The same discussion, just with the order of the pairs reversed applies to the "member of" connection. Thus "member of" is also an implicit relation. For examples of structural relations see Section 6.

5 Connections between Object Classes

We shall show now that both semantic connections between classes, "role of" and "category of" are also implicit relations. However, there is a difference from the previously discussed implicit relations in that these are defined between the extensions of the related object classes rather than between the domains of the related object types as for subtype, set of, and member of.

So far we have assumed that any two object classes model different objects of the real world. But if we want to model the same real world object in two different contexts, in which the objects have different realizations, we must introduce two different object classes. Each object class has a different object type. To capture the semantic connection between the two classes we introduced the role of concept. It is used to express the fact, that two (or more) object classes model the same real world object in different contexts (or equivalently the objects are represented in different roles). This concept has consequences regarding message passing, because if a method is sent to an object and the method is not contained in the interface of that object, then it may be forwarded to one of its roles. More details are given in [NPGT89a, NPGT89b]. The object types of the classes A and B do not have to be sub- or supertype of each other. The role of concept reflects purely a semantic connection, which is not reflected by a relation between the corresponding object types.

The second modeling device for relating object classes is called category of. In some way it is dual to the role of construct. It is used to model the same real world object with additional knowledge, but still in the same context. Thus it is a refinement of the description with respect to one aspect of its former description. Whereas in the case of role of the additional information was about the object in a different context. If more specialised information about the instances of an object class is available, we can categorise the instances into different object classes and relate these classes via the category of concept with the original class.

Note that by the definition of "A is category of B" the object class A represents the same real world objects represented by the object class B in the same context but with some extra information describing the refinement. In such a case the object type A' must contain all the properties of the object type B' and some additional ones to reflect the more refined information known in the object class A. This is the case if the object type A' is a subtype of the object type B'. Hence, if the object class A is a category of an object class B, then the corresponding object type A' must be a subtype of the object type B'.

However, if the object class A is category of object class B, then the instances of the class A model the same real world objects as the class B. This has to be seen in contrast to two object classes in which the corresponding object types are in a sub-/supertype relation. There the two object classes model objects which are structurally closely related, but they may correspond to different real world objects from dissimilar areas.

The set of object classes forms with respect to the role of and category of specifications a network [NPGT89a, NPGT89b]. This network is defined independently from the hierarchy of the object types. Thus, the object types of two object classes modeling the same real world objects need no longer be in a sub-supertype relation. In previous data models such a relation always implied a structural similarity of the instances. This is the case in the situation of category of specialization, but certainly not in the case of role of specialization. There, the same real world object can have totally different structures in different roles. By separating the semantic specification from the definition of the object types, our model is closer to the real world. For the discussion of inheritance between semantically connected classes see [NPGT89a,NPGT89b].

Finally we want to show that both, role of and category of are implicit relations. If class A is role of class B, then both classes describe the same real world object, but in two different contexts. Thus the role of connection from A to B can be described as a relation containing pairs (a, b) such that a is an instance of A, b is an instance of B and a and b describe the same real world object. This relation is implicit since the pairs do not need to be listed, but are determined by the mathematical identity relation. This identity relation is obtained by transitivity from the identity relation from an instance of A to the corresponding real world object and the identity relation from this real world object to the corresponding instance of B.

If class A is category of class B then both classes describe the same real world object in the same context, but A describes it with additional knowledge. As for the role of relation this implies that category of is an implicit relation where the mathematical identity relation, obtained as for the role of relation, is used to determine the pairs of the relation.

6 Example

We describe an example from a manufacturing environment to demonstrate the use of the different elements in our model. The example is basically self explanatory. We just highlight some important points.

In the following the names of the object classes are spelled with small letters, object types with capital letters, and keywords are in bold face. The two columns describe the corresponding object types (left) and classes (right).

We omit relationships and methods from the class descriptions whenever there exists only one class for a given object type, even though it is in principle necessary to specify them. Thus we write the classes for relationships and methods only for object types which have multiple classes.

```
objecttype PERSON               class person
  attributes:                     objecttype: PERSON
    Name: STRING                  essential: Name, Address
    Address: STRING

objecttype CUSTOMER             class customer
  subtypeof: PERSON               objecttype: CUSTOMER
  attributes:                     roleof: PERSON
    Creditline: INTEGER           essential: Creditline
  relationships:
    Residentin: REGION

                                class formercustomer
                                  objecttype: CUSTOMER
                                  roleof: person



objecttype SALESPERSON          class salesperson
  subtypeof: PERSON               objecttype: SALESPERSON
  attributes:                     roleof: person
    SocialSecNo: INTEGER          essential: SocialSecNo
    Salary: INTEGER
  relationships:
    Regioncovered: REGION

objecttype REGION               class region
  attributes:                     objecttype: REGION
    Name: STRING                  essential: Name
    Geographicarea: STRING
  relationships:
    Responsiblesalesperson:
    SALESPERSON

objecttype PRODUCT              class product
  attributes:                     objecttype: PRODUCT
    Productno: INTEGER            essential: Productno
    Manufacturer: STRING
    Price: REAL
  relationships:
    Orderby: ORDERS
```

```
methods:
  Orderingcustomernames:():
  Orderby → ORDERS:
  setof → {ORDER}:
  Customername → {STRING}
  Totalquantity: ()
  Orderby → ORDERS:
  setof → {ORDER}:
  Quantity → {INTEGER}:
  Sum → INTEGER


objecttype                        class  compoundproduct
  COMPOUNDPRODUCT                   categoryof:  product
  subtypeof:  PRODUCT
  attributes:
    Noparts: INTEGER
    Assemblytime: INTEGER
    Assemblycost: REAL
  relationships:
    Parts: PRODUCTS

objecttype  PRODUCTS              class  products
  setof:  PRODUCT                   objecttype: PRODUCTS
  attributes:
    Noproducts: INTEGER

objecttype  ORDERS                class  orders
  setof:  ORDER                     objecttype:  ORDERS
  attributes:                     dependon:  product
    Noorders: INTEGER

objecttype  ORDER                 class  order
  memberof:  ORDERS                 objecttype:  ORDER
  attributes:                       dependon:  orders
    Orderdate: DATE                 relationships:
    Quantity: INTEGER                 Orderingcustomer: customer
  relationships:
    Orderingcustomer:
    CUSTOMER
  methods:                        methods:
    Customername: ():               Customername: ():
    Orderingcustomer →              Orderingcustomer → customer:
    CUSTOMER:
    Name → STRING                   Name → STRING
```

Note that the object type CUSTOMER is used by two classes, customer and formercustomer. They differ in that only in customer the attribute "creditline" is essential. Customer, formercustomer and salesperson are role of person. Note that a salesperson can also be a customer.

We define an object type and class compoundproduct for describing a complex product which is assembled from simple products as a subtype of PRODUCT and category of product.

The set of all orders of a given product is described in ORDERS which is defined as a set of ORDER representing a single order. Orders are dependent on product and order in turn is dependent on orders (for which it is a member of) and thus indirectly dependent on product because dependency is transitive. Clearly, no order is possible for a product which does not exist.

The pair of parentheses following the name of a method stands for the object type in which the method is defined. The method chain is described in triples of *property → object type:*, meaning that the property applied to the object type at the end of the previous triple yields the object type of the current triple. The colon is used to separate two triples. In the triples of the transformer chain the *object type* is replaced by a *data type*.

The method Customername in ORDER finds the name of a customer of a given order by using a relationship Orderingcustomer to CUSTOMER for which the attribute Name is defined. In the class description we specify that CUSTOMER refers to class customer and not class formercustomer for both the relationship Orderingcustomer and the method Customername.

Similarly two methods are defined for product. Totalquantity finds how many copies of a given product are ordered; it consists of a method chain composed with a transformer chain. The method chain starts with the relationship Orderby defined for product, giving ORDERS. Applying the setofof relation gives a set of ORDER, i.e. {ORDER}. At this point the transformer chain starts. Applying the attribute Quantity gives a set of integers whose sum is the required result. The operation Sum takes a set of integers as argument and is defined in the underlying programming language. The method Orderingcustomername is similar, only the final result is not one element but a set of names.

7 Conclusions

This paper was motivated by our understanding that in order to have effective integration of different systems the elements of those systems should be well defined. If one attempts to integrate systems which are not well defined or not well understood, problems tend to multiply and cause mistakes, inefficiencies, and misunderstandings.

In this paper we thus undertake a twofold task. First we present the Dual Model which determines for each part in the definition of an object class whether it is structural or contains semantic information. Secondly, we provide theoretical definitions for the elements of the Dual Model. Such a model will support simpler, easier and more effective integration of systems.

Thus we have introduced a new distinction in the specification of types for object-oriented data and knowledge bases. We have divided the traditional data representation based on class hierarchies into two different hierarchies, a structural hierarchy of object types and a network of object classes. Our reason has been that traditionally two classes that are intimately connected by a semantic relationship must also look structurally similar. This condition seems practically too limiting and theoretically not justified. We have termed the model created by this new distinction between structural and semantic elements the "Dual Model" for Object-Oriented Knowledge Bases.

An object type may contain attributes, relationships, and methods. Object types can be related to each other by user defined relationships as well as the special structural relations subtype of, member of, and set of. Object classes are linked to their corresponding object types, and one object type may function as a structural template for several unrelated object classes. Object classes in turn may be connected by the above mentioned user defined relationships as well as the special semantic relations category of and role of. Relationships may be marked as essential, and classes may be made dependent on other classes. These two specifications also express semantic constraints and are therefore specified with an object class rather than its corresponding object type.

The classes for relationships and methods need to be specified individually for every given object type, however if there is only one object class for a given object type the obvious default assumption will be made.

The existence of two kinds of hierarchies each of which consists of different kinds of structures leads to interesting issues of inheritance which have been discussed in this paper [NPGT89a,NPGT89b].

Furthermore for each element in the definition of a class in the Dual Model, namely attributes, relationships, and methods, we have identified a proper mathematical underlying structure. In particular we show that the structural connections subtype of, member of, and set of, as well as the semantic connections role of and category of are all mathematical relations of a kind which we call implicit relations.

Thus, all parts of our Dual Model are well defined, both from the aspect of their structural or semantic nature and from the mathematical aspect. Therefore, the Dual Model will function as a good basis for the integration of different systems.

REFERENCES

[AM87] Andrews, T. and Morris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment", Proceedings of the OOPSLA Conference, 1987.

[CM84] Copeland G. and Maier, D., "Making Smalltalk a Database System", ACM SIGMOD, June 1984.

[DKT88] Duchene H., Kaul, M., Turau, V., "Vodak Kernel Data Model", in K.R.Dittrich (Edt.), Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science, No. 334, 1988.

[DN66] Dahl, O.J. and Nygaard K., "Simula - an ALGOL based simulation language", Communications of the ACM, No. 9, 1966.

[DT88] Danforth S., Tomlinson, C., "Type Theories and Object-Oriented Programming", ACM Computing Surveys, Vol 20, No. 1, 1988, pp. 29 - 72.

[F87] Fishman, D. et al., "IRIS: An Object-Oriented DBMS", ACM Transactions on Office Information Systems, 4(2), April 1987.

[FKRST89] Fischer D., Klas, W., Rostek, L., Schiel, U., Turau, V., "VML - The VODAK Data Modelling Language", GMD-IPSI, Technical Report, Dec. 1989.

[GR83] Goldberg, A. and Robson, D., "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading Massachusetts, 1983.

[KNS88] Klas W., Neuhold, E. J., Schrefl, M., "On an Object-oriented Data Model for a Knowledge Base", in: R.Speth (Edt.), Research into Networks and Distributed Applications - EUTECO 88, North-Holland, 1988.

[LR88] Lecluse, C. and Richard, P., "Modeling Inheritance and Genericity in Object-Oriented Databases", LNCS #326, ICOT 1988, p. 223-237.

[MSOP86] D.Maier, J.Stein, A.Otis, A.Purdy, "Development of an Object-Orienred DBMS", Proc. of 1st Int. Conf. on OOPSLA, Portland (Oregon), October 1986.

[NK89] Neuhold, E. J., Kracker, M., "Extending Knowledge Craft with a Persistent Store by using the VODAK database system", GMD-IPSI Technical Report, 1989, submitted for publication.

[NPGT89a] Neuhold, E. J., Perl, Y., Geller, J., Turau, V., "Separating Structural and Semantic Elements in Object-Oriented Knowledge Bases", Advanced Database System Symposium, Kyoto, Japan, 1989.

[NPGT89b] E. Neuhold, Y. Perl, J. Geller, V. Turau, "The Dual Model for Object Oriented Knowledge Bases", New Jersey Institute of Technology, Tech Report CIS-89-23, submitted for publication.

[PM88] Peckham J., Maryanski, F., "Semantic Data Models", ACM Computing Surveys, Vol 20, No. 3, 1988, pp. 153 - 190.

[NS88] Neuhold, E.J and Schrefl, M., "Dynamic derivation of personalized views", Proceedings of the 14th International Conference on Very Large Data Bases, Long Beach, CA, 1988

[SN88a] Schrefl, M., Neuhold, E. J., "Object Class Definition by Generalization Using Upward Inheritance", Proceedings of the 4th International Conference on Data Engineering, Los Angeles, CA, 1988, pp. 4-13.

[SN88b] Schrefl, M., Neuhold, E. J., "A Knowledge-Based Approach to Overcome Structural Differences in Object-Oriented Data Base Integration", Proceedings of the IFIP Conference on The Role of Artificial Intelligence in Data Bases and Information Systems, Canton, China; North Holland Publishing Company, 1988.

[SR86] Stonebraker M., Rowe, L., "The Design of POSTGRES", in Proc. of ACM SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[S86] Stroustrup, B., "An Overview of C++", SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 7-18.

[W87] Wegner, P., "The Object Oriented Classification Paradigm", Research Directions in Object Oriented Programming, ed. Schiver, and Wegner, MIT Press 1987.