Dynamic Derivation of Personalized Views[1]

Erich J. Neuhold and Michael Schrefl[2]

Institute for Integrated Publication
and Information Systems
GMD, Darmstadt, FRG

Abstract

Traditionally user interfaces to databases have either assumed complete knowledge of the conceptual schema of the database or they have relied on the utilization of predefined views to restrict the universe of discourse for end users or application programmers. This paper introduces an interface mechanism to dynamically derive personalized views and consequently a dynamic learning facility for the utilization of a database. The principles are based on ideas developed for the dynamic creation of universal views in multi-database environments.

1. Introduction

In database applications we usually can assume that end users and application programmers only need to know some particular subset of the whole conceptual schema and consequently the database. The traditional answer to this situation has been to provide users or user groups with their specific views on the conceptual schema. These user views are completely predefined by the database administrator, and the users can pose queries and (restricted) updates only against these views.

This approach is appropriate if the scope of the queries a user may want to formulate is known in advance. But in case a user wants to pose an unexpected query, or in case a new user wants to access the database a predefined view is not the answer. To solve the problem two solutions have been proposed: (a) a new user view is defined before the query is actually posed, or (b) the user must learn about the whole conceptual database schema and pose his/her query accordingly. In order to support the second alternative recent research in database systems has investigated graphical tools for browsing through the objects of a database [KM84, GK85, MOT86, BH86, LAR86].

Our research combines both alternatives, and extends them to the framework of object oriented systems [BAN87,GR83,DS86,COX86]. The following scenario is assumed: The user is allowed to formulate his queries against a hypothetic view of the database. A hypothetic view exists only in the mind of the user and has not yet been actually established. An intelligent knowledge-based system will assist the user to materialize this hypothetic view when the queries are processed. The user hereby is freed to a large extent from learning about the sometimes very complex database schema and from navigating on the logical level through that schema.

Our work in the framework of object-oriented systems most readily compares to the work on universal schema interfaces to relational databases. In a universal schema interface [MAI83, MRSS87] accesses to the database are formulated using attributes only. It is assumed that the relationship among these attributes can be automatically derived by joining several stored relations. In order to ensure correct and meaningful answers two requirements have to be satisfied: First, every attribute must have exactly one meaning, i.e it may not be semantically overloaded (unique role assumption). Secondly, the sets of attributes with a meaningful semantic relationship have to be predefined as "objects". If a universal relation is assumed the set of maximal objects can be computed from functional and multivalued dependencies [MU83].

The implicit join method [LIT85] has been developed as an alternative to the universal schema approach. Again the user is allowed to omit interrelational joins in his query. But, contrary to the universal schema approach no objects have to be predefined. The necessary joins are derived from natural dependencies among the attributes.

1) This research was partially supported by the Austrian Science Foundation ("Fonds zur Foerderung der wissenschaftlichen Forschung") under Project No. P5976P

2) Current Address: Institut fuer Angewandte Informatik, Technical University of Vienna, Austria

A natural dependency exists between two attributes if they share a domain and at least one of them belongs to the primary key. Incomplete queries are completed by means of a query graph. The nodes correspond to relations, the edges to interrelational joins. Incomplete queries have an unconnected query graph. They are completed by adding as few natural dependencies as possible to connect the query graph. However, the minimization of the number of joins may not always be the correct interpretation of a user query. The number of joins is merely a syntactic measure. As an alternative we suggest to use an object-oriented semantic data model and a knowledge-based approach to complete incomplete queries. The research is based on concepts used for navigation free queries in the relational model, on research in the field of semantic and object-oriented data modelling [MBW80,SHI81,HM81,NEU86], and on research in object-oriented database integration [SN88a, SN88b].

An introduction to the object-oriented database model used within our research is presented in Section 2. The overall approach to derive personalized views is outlined in Section 3. In Section 4 the principles of navigating in the conceptual schema are discussed. This is followed in Section 5 with the handling of complex queries during the construction of the personalized user views. Finally, conclusions and future research directions are presented in Section 6.

2. The object-oriented database model

The VODAK-data model will be used as the basis of the discussions. The model is a synthesis of semantic data modelling concepts and object-oriented system principles. A more detailed description of the model can be found in [KNS88]. The model includes the four main abstraction principles of semantic data models classification, aggregation [SS77], specialization, and grouping [HM81]. (The latter will not be discussed here).

Classification means that objects having the same type of properties are collected to object classes. The properties of an object are represented by attributes and by relationships. An attribute takes an instance of a printable data type as its value. In contrast, relationships take non-printable object identifiers as values. For example, in the ORDER database of Figure 1 the class CUSTOMER has the attributes Name, Address and Credit, and the relationship ResidentIn, which identifies the REGION in which the customer resides. The object class REGION in turn has the relationship ResponsibleSalesman, which specifies the salesman who is responsible for all customers of the region.

The other abstraction principles are expressed by different semantic relationship types introduced in the following.

Aggregation is supported in two senses: (a) to model a relationship in which several independent objects participate, and (b) to model dependent components of a superordinate or complex object. The first case is expressed by a has-constituent relationship, the second case by a has-component relationship. Their inverses are constituent-of and component-of. A specific name may be given to a particular occurrence of a relationship type. For example in Figure 1 the class SHIPMENT-OFFER models the fact that carriers offer to ship a product to some region at a specific price. The three has-constituent relationships to the classes CARRIER, PRODUCT, and REGION have been named ShippingCarrier, ShippedProduct and DeliveredIn. The price of a particular shipment is represented by the attribute ShippingCharge.

Specialization is used (a) to classify real word objects which are modelled by a superclass into disjoint subclasses (category-specialization), and (b) to model real world objects in different real world situations (role-specialization). Contrary to the principle of type specialization in object-oriented programming languages [GR83, COX86] a real world object which is represented by an instance of a subclass is also represented by an instance of the superclass. The instance of the subclass and the instance of the superclass, which model the same real world object, are related by the role-(specialization)-of, rsp. category-specialization-of relationship. Its inverses are has-role and has-category-specialization.

In the example of Figure 1 the class PRODUCT has the set-valued component OrderedBy. An instance of the class ORDERING-CUSTOMER represents a CUSTOMER in the role of being the customer of a particular product. As such he/she has ordered the particular product on a certain date in a certain quantity. This is expressed by the attributes OrderDate and Quantity. Note: if some customer has ordered several products he/she will be represented by several instances of the class ORDERING-CUSTOMER, but only by a single instance of the class CUSTOMER.

When modelling one has to be careful to identify the dependencies between the object class instances (which will be simply refered to as objects in our further discussions): A relationship object (aggregate object) depends on the objects which constitute the relationship; a component object depends on its superordinate object; and an instance of a subclass is dependent on the instance of the superclass which models the same real world object. Therefore an object is directly dependent on another object if one of the following semantic relationships holds between them: has-constituent, component-of, role-of, and category-specialization-of.

```
class CUSTOMER
   attributes:
      Name:       STRING
      Address:    STRING
      Credit:     DM
   relationships:
      ResidentIn:     REGION
end CUSTOMER

class REGION
   .....
   relationships:
      ResponsibleSalesman: SALESMAN
end REGION

class SHIPMENT-OFFER
   has-constituents:
      ShippedProduct:      PRODUCT
      DeliveredIn:         REGION
      ShippingCarrier:     CARRIER
   attributes:
      ShippingCharge:      DM
end SHIPMENT-OFFER

class PRODUCT
   attributes:
      ProductNo:           INTEGER
      ManufacturedFirst:  YEAR
   methods:
      Price:      DM
   has-components:
      OrderedBy:set-of ORDERING-CUSTOMER
end PRODUCT

class ORDERING-CUSTOMER
   component-of: PRODUCT
   role-of: CUSTOMER
   attributes:
      OrderDate:      DATE
      Quantity:       INTEGER
end ORDERING-CUSTOMER

SALESMAN and CARRIER need not be
further specified here.
```

Figure 1: Order database

These semantic data model concepts are merged with object-oriented system concepts. The fundamental principles of object-oriented systems are data abstraction and inheritance. Data abstraction means that a data structure and its associated operations are defined together and encapsulated into an abstract data type, or abstract module. In object-oriented systems the abstract module is called an object class and will be considered equivalent to an object class of the semantic data model. The operations are defined for the instances of the class and are called methods. A method is executed for an object by sending the object a message which identifies the method by a method-selector and which possibly carries several actual parameters. Such a message will be denoted as: [<addressee> <message-selector>: {<argument-values>}], where the addressee is either an object class, or a variable containing a set of instances. In the latter case the message will be sent to all of them. A mnemnonic identifier enclosed in single quotes will be used to denote that a message is sent to a particular instance, which is actually identified by an unprintable internal instance identifier.

We assume that an attribute value may be retrieved by using the attribute name as a message selector. One or more objects related to an object by some relationship may either be retrieved by a message, using the name of the relationship as message selector, or by a message, using the name of the type of semantic relationship as message selector and the name of the related class as actual parameter. For example the customers of the product 'product-5' can be either retrieved by ['product-5' OrderedBy] or by ['product-5' has-component: ORDERING-CUSTOMER].

Furthermore we will assume that the message "where:<condition>" sent to a class, or a variable containing several instances, will return those instances of the class, or variable, which satisfy the given condition.

Inheritance is utilized as follows: If an instance of a subclass can not handle a message, the message is delegated to the instance of the superclass, to which the first instance is related by a role-of or category-specialization-of relationship.

3. The overall approach to derive personalized views

We believe that due to missing semantics an intelligent user interface should not be built directly on a conventional database system. Rather a knowledge based model should be provided as interface to a given relational, fact, or document database. This knowledge base is initially established by some domain expert together with the database administrator, for those parts of the database that are thought to be needed in the specific application domains. Some initial research results on the interactive process to develop this knowledge base may be found in [NS88]. It is an object oriented and semantically enriched representation of the original database schema. For example, domains are provided for all attributes; object classes, attributes, relationships, etc., are identified, and synonymous names are added to the original names. Specialized operations on these databases are represented as methods attached to the object-classes. This initial knowledge base will be continuously refined. Such a refinement will be necessary whenever the knowledge base has been found to be incomplete, or in case the conceptual schema of the original database has been changed.

Whenever an end user or application programmer wants to utilize the database he/she has some "objects of interest" in mind. These objects of interest will then be in the center of the user's attention toward the database for at least the duration of several queries, establishing in that way a user's view on the database. However, at first this view is purely hypothetical, that is it only exists in the mind of the user, and will have to be established in reality through interactions between the user and the knowledge base. These interactions and the actions in the knowledge base will be discussed in the remainder of this paper.

The user poses a simple query by sending a message based on his "hypothetical" object of interest. In the user's hypothetical view of the database the object is assumed to have a method which implements the response to the message. But the object and the identified method may not yet exist in his "real" view and moreover the object might actually not have been defined with the referred method. In this case a new view component, which is compatible with the user's hypothetical view, must be derived dynamically.

For example the customer Smith may be the object of interest in the query posed by some salesman. The salesman will now assume that the object 'Smith' can respond to the message "OrderedProducts" by delivering to him the set of products Smith has ordered. But investigating the Order database of Figure 1 it becomes clear that these products must be retrieved by selecting those instances of the class PRODUCT which contain in the component OrderedBy an instance which represents Smith in the role of customer of some product.

Before we develop the concepts to derive a view dynamically, let us analyze the reasons why the personal perspective of some user may not directly match the objects and methods in the conceptual schema. In the simplest case a user may want to address a data model concept, which is present, but has been given a different name by the database administrator and/or the domain expert. More complicated, a user may assume that some real world fact is represented by a certain data model construct, but actually it has been modelled by a different one, e.g., the customer situation illustrated above.

Differences in naming can be resolved by a knowledge navigator mechanism which uses domain specific thesauri in order to find the appropriate synonym. To handle structural differences is more cumbersome. In this case a message which is received by some object cannot be handled by the object. Instead, the message has to be answered by some semantically related object(s). Our approach to handle structural differences can now be characterized as follows: In a first step, the objects which can at least respond to an adapted form of the message have to be identified. Then the original message has to be appropriately transformed and forwarded to these objects. Finally, the answers from the various objects have to be combined.

As a simple example consider the message "ResponsibleSalesman" sent to a specific customer. No appropriate method can be found with the class CUSTOMER, because salesmen are only assigned to regions and not to specific customers. Nevertheless a salesman is responsible for all customers that reside in his region. If therefore the original message is forwarded along the relationship "ResidentIn" to the customer's region the appropriate method will be found and the expected answer can be returned.

A <u>message forwarding plan</u> defines at the class level how messages that cannot be handled directly are to be forwarded to semantically related objects. For example, the message forwarding plan

a = "ResidentIn REGION ResponsibleSalesman SALESMAN"

will be associated, using the selector "ResponsibleSalesman", with the class CUSTOMER in the created customer view. As another example, consider the message forwarding plan for OrderedProducts (see Figure 2)

b = "has-role ORDERING-CUSTOMER component-of PRODUCT"

which states how products ordered by a specific customer can be determined. Another message forwarding plan

c = "((component-of PRODUCT constituent-of SHIPMENT-OFFER) intersect (role-of CUSTOMER ResidentIn REGION constituent-of SHIPMENT-OFFER)) has-constituent CARRIER"

states that the carriers an ordering customer may possibly use must accept the product ordered for shipping, and that they must ship this ordered product into the region where the customer resides. A message which follows the first path via PRODUCT will retrieve the carriers who are able to ship the product the customer has ordered. A message which follows the second path via REGION will retrieve the carriers who offer a shipment into the region where the ordering customer resides. The intersection of the set of carriers returned by the two messages produces those that may possibly serve the ordering customer.

The message forwarding plans a and b will eventually be incorporated into the customer view illustrated in Figure 2, where object-coloring and upward inheritance as explained in [SN88a] provide for the other properties of the object class CUSTOMER-V: Instances can be tagged (colored) with a class-name to tell them to follow primarily the behavior of that class. For example if the instance 'Smith' of CUSTOMER is colored with CUSTOMER-V, i.e., 'Smith$^{\text{CUSTOMER-V}}$', the message "Credit" will retrieve his credit in US$ instead of DM. Properties not defined at the view class CUSTOMER-V are inherited upward from the class CUSTOMER.

```
class CUSTOMER-V
   view of: CUSTOMER
   attributes:
      Credit:    US$
   methods:
      ResponsibleSalesman: SALESMAN
         message forwarding plan
            ResidentIn REGION
            ResponsibleSalesman SALESMAN
      OrderedProducts: PRODUCTS
         message forwarding plan
            has-role ORDERING-CUSTOMER
            component-of PRODUCT
end CUSTOMER-V
```

Figure 2: Personal view CUSTOMER-V

4. The concept of message forwarding

When developing a message forwarding plan we employ principles similar to those adopted for the dynamic integration of object-oriented databases [SN88b]. For reasons of code sharing the message forwarding plan is not developed at the instance level but rather at the class level. In order to produce a plan two steps have to be taken:

(a) Path detection: An inquiry message is initiated and forwarded along "promising" semantic relationships to other object classes in the conceptual schema. To select "promising" relationships close cooperation with the interactive knowledge navigator is required. The knowledge navigator, an expert system which uses a domain knowledge base, various thesauri, and general knowledge components about the user and his environment, will not be discussed here. However, if the knowledge navigator fails, the system will seek the help of the requesting user, the domain expert, or even the data base administrator to identify the necessary semantic relationships, object classes, and methods.

(b) Path combination: From the possible paths determined during the path detection phase those of semantic relevance will be selected and combined. Contrary to other approaches, this selection is not based on the physical length of the path, but on the semantic surroundings of the object classes visited on the path. The concept of context of an object tries to capture the informal notion of these object environments by the definition: The context of an object o is the set of objects on which o is dependent. The context of an object class O is the set of object classes which contain the instances on which the instances of the class O are dependent.

In the following the algorithms necessary for path detection and path combination are discussed and some of the principles (rules) are exemplified. A complete set of rules to be followed by the algorithm is given in the appendix of the paper.

Message path detection algorithm:

When a message is forwarded from an object o1 to another object o2 the following situations may exist:
(a) context (o1) > context (o2)
(b) context (o1) < context (o2)
(c) otherwise

In case (a) the new context contains only a subset of the objects of the previous context. This occurs when the new object o2 belongs to a more general object class O2, that is, it is less specified and therefore depends on fewer objects than the instance o1 of the class O1. The trivial situation exists when O2 is a superclass of O1. We then say that a context generalization occurs when a message is forwarded from o1 to o2.

In case (b) the new context contains more objects than the previous context. This situation arises when the object o2 belongs to a more specific class O2, that is, it is specified in more detail than the object o1 of class O1 and therefore depends on more objects. In the trivial situation O2 is a subclass of O1. However, considered globally we have to distinguish whether the message originally has been passed from a specialization of O1, a generalization of O1, or some other context. If the message has come from some more specific context of O1, e.g., some subclass of O1, which of course has to be different from O2, then considered in total a context switch occurs. In the other situations a context specialization occurs when a message is forwarded from o1 to o2.

In case (c) a context switch will always occur.

Context specializations and generalizations can be handled automatically by the system. They correspond to key joins resp. to projections in relational databases and a connection trap [COD70] cannot occur. Differently, a context switch corresponds to more general joins where a connection trap is possible and consequently the user, DBA, or domain expert has to be involved to clarify the semantic situation that exists.

An inquiry message is not forwarded further, if:
(1) the method searched for has been located, or
(2) a cycle has been encountered, or
(3) a context left previously is reentered, or
(4) too many context switches occurred.

A cycle is found when an object class already encountered during forwarding is entered again. Here we have to distinguish between two cases. First, the cycle corresponds to a possible recursive query, e.g., a recursion over the part-subpart hierarchy "PART has-category-specialization COMPLEX-PART has-component SUBPART role-of PART", or secondly, it is of some type which is not semantically meaningful. In the first case the cycle will be added to the possible paths, but in both cases a cycle will not be followed again.

If an inquiry message reenters a context it previously left the forwarding process can also be terminated, as further forwarding will not add new information to the detection process.

A context switch always carries the user from one environment to a new one. Each time he has to learn to understand the new environment in order to make the necessary decisions on the acceptability of the identified paths. With too many such switches it is very likely that the user will be too confused to make reasonable decisions, and we feel it is then better to terminate the search. This context switch based approach is semantically much more appropriate than to limit the search to a maximum number of forwarding steps, as it is done in other solutions to the problem.

Message plan combination algorithm

To create the final message forwarding plan the answers gained from the path detection algorithm have to be combined. The algorithm chosen for message path combination again is derived from the algorithm developed for the dynamic creation of global views in multi-database environments [SN88b].

Message forwarding plans that preserve the (local) context are favoured against others. Context switches in a plan will always be pointed out to the requesting user for approval.

In principle two message forwarding plans are candidates for combination if they intersect at some class. Here again plans which do not intersect with any other plan before they intersect with each other are preferable, because their front parts may be combined into a single plan.

The algorithm therefore contains the following steps:

1. Build all equivalence classes P_C of plans that intersect for the first time with object class C. Note: that a common prefix of two paths is not considered an intersection.

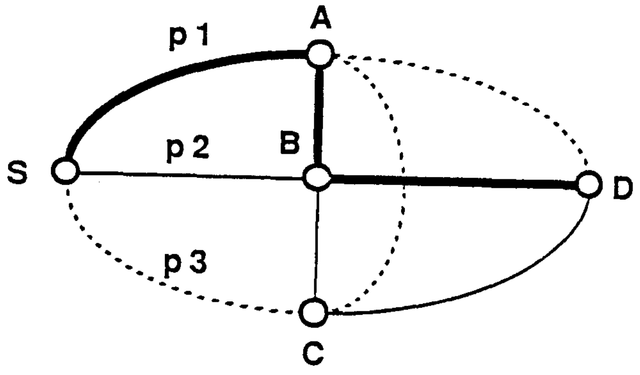
2. From every equivalence class P_C choose the two plans which have the minimum number of context switches between the last object class in a common prefix and the object class C.

3. Combine the selected plans from an equivalent class P_C
(a) by "set intersection" if C is reached from a context generalization in every inherent path of the two plans,
(b) by "set union" if C is reached from a context specializationin every inherent path of the two plans,
(c) by an operator supplied by the user otherwise.

4. Reintegrate the new (combined) plan into the equivalence classes and repeat steps 2 and 3 until no further plans can be combined.

5. In case some equivalence classes are circularly dependent on each other the plan combination steps 2,3, and 4 will not result in a single (combined plan) and special steps have to be taken to resolve the circular dependencies as we will show by an example. (The complete rules can be found in the Appendix).

Assume three plans p1, p2, and p3 and the three corresponding equivalence classes P_A, P_B, and P_C. The circular dependencies now could have the form



where l1 = length of p1 between A and B
l2 = length of p2 between B and C
l3 = length of p3 between C and A
and l2 > l1 > l3.

We select now the path where li is maximum, i.e., p2, and replace the subpath from S to C via B by the direct subpath of p3 from S to C. That is, we form a common prefix between p2 and p3 breaking the circular dependency.

6. After step 5 reintegrate the new (combined) path into the equivalence classes and repeat steps 2 to 6 until a single (combined) plan remains.

Using our Order database in Figure 1 as an example, the inquiry message for "Carrier" initiated at the class ORDERING-CUSTOMER will return with the paths

p = "component-of PRODUCT constituent-of SHIPMENT-OFFER has-constituent CARRIER"

and

q = "role-of CUSTOMER ResidentIn REGION constituent-of SHIPMENT-OFFER has-constituent CARRIER".

The plans p and q intersect for the first time at the class SHIPMENT-OFFER. The plans reach the class SHIPMENT-OFFER, i.e., the context {SHIPMENT-OFFER, CARRIER, PRODUCT, REGION} from different context generalizations, i.e., {PRODUCT} and {REGION}. Therefore they are combined using an intersection. This results in the plan c given above. Using an intersection ensures that a carrier returned by the final plan actually ships the product which the customer has ordered into the region in which the customer resides.

5. Complex queries

So far we have not treated cycles that were encountered by the path detection algorithm. After a single (combined) message forwarding plan has been produced by the steps 1 to 6 above, the cycles have to be investigated and possibly integrated into the plan. A detailed discussion is beyond the scope of this paper but we have included the rules used by the integration algortihm into the appendix. The rules essentially ensure that only "useful" cycles are included in the plan., and rule 6 given there furthermore ensures that only cycles are included which can be expected to terminate during the actual execution of a query against the plan.

For example a message forwarding plan representing a part explosion problem, e.g. collecting the combined weight of a complex part by the plan

> "(has-category-specialization COMPLEX-PART has-component SUBPART role-of PART)* has-category-specialization SIMPLE-PART Weight KILO"

developed for the class PART, will terminate, as every cycle will eventually end with an instance of a SIMPLE-PART.

Thus far, we have investigated the situation where an end user or application programmer has a specific "object of interest" in mind and builds for himself a personalized view concerning the context of this "object of interest", with the methods and algorithms described in Sections 3 and 4. The mechanisms will normally produce an object class hierarchy with the class of the object of interest as the root and the other object classes either below or connected to it through relationships.

An object class below the root may itself have two kinds of methods and attributes attached:
(a) methods and attributes which model properties that are independent from the specific object of interest the user has in mind,
(b) methods and attributes which model properties that are dependent on the object of interest.

With the concepts presented so far an object retrieved by a forwarded message will only be able to have methods and attributes of type (a) above, as the relationships between the different objects touched by the forwarded message will be lost from one query execution to the next.

However, situations in which messages can refer to the environment of the preceeding message(s) are quite natural in a database. For example consider the Order database of Figure 1 and the message ordering-customers:= ['prod632' has-component: ORDERING-CUSTOMER], which retrieves the customers who have ordered the product numbered "632". The variable ordering-customers will then contain the customers in the role of customers of the product "632". Therefore, the date on which customer Smith has ordered product "632" can be retrieved by [[ordering-customers where: Name = "Smith"] OrderDate]. As it is implicitely assumed that the message OrderDate will retrieve the date on which customer Smith and not some other customer has ordered product "632" the message refers to the environment of the previous message [has-component: ORDERING-CUSTOMER].

In Section 3 we have explained how a user view class CUSTOMER-V (Figure 2) can be constructed and how the products ordered by a specific customer may be retrieved into a variable, e.g. ordered-products, by the message ordered-products:= ['Smith$^{\text{CUSTOMER}}$, OrderedProducts]. Now assume that the OrderDate of the product "632" ordered by Mr. Smith is to be retrieved. It would be expediant if, in analogy to the above, the message [[ordered-products where: ProductNo = "632"] OrderDate] could be used. Here the message OrderDate refers to the environment of the previous message OrderedProducts as we implicitely assume that the date should be returned on which the customer Smith, and not some other customer, has ordered product "632". Now let us investigate what actually will happen to the message ['prod632' OrderDate], which is one of the messages produced by the above request, if it is treated unconnected to any previous message. As the class PRODUCT to which the instance 'prod632' belongs has no appropriate method for OrderDate a message forwarding plan will be initiated. The plan which will be suggested to the user will be "has-component ORDERING-CUSTOMER OrderDate DATE". But this plan will retrieve not only the date when Mr. Smith has ordered the part "632" but also the dates when other customers have ordered the part.

The message OrderDate sent to the product 'prod632' does not deliver the desired result because the environment in which the product originally has been retrieved has not been kept with the object identifier 'prod632'. In contrast, we have seen that the message OrderDate sent to the instance of ORDERING-CUSTOMER which represents Mr. Smith in the role of a customer of the product "632", i.e., 'Smith-ordering632', retrieves the desired result as this instance reflects the appropriate environment. As a solution to our problem, we will retain with any object returned by a forwarded message the environment(s) through which the object has been retrieved. For this purpose the object is context colored with the most specific environments through which the message has been forwarded.

A simplistic approach would be to color the object by all the objects the message has visited in its forwarding process. However, it is not necessary to represent all objects visited by the message in the context-color. It will be sufficient just to remember the objects which determine the most specific environments (contexts) the forwarded message has visited. In the following we analyse how these objects can be determined.

A class C represents a most specific context in a message forwarding path, iff its context is a superset of the context of its predecessor and of the context of its successor in the path, providing they exist.

Similarly, we want to identify those classes of a message forwarding plan whose instances identify a most specific context in any execution of the plan. In addition, we would like to determine the most specific contexts of a message forwarding plan statically and not dynamically during the execution of the plan.

If two inherent paths of a message forwarding plan intersect with each other at a class which represent a most specific context in both paths, then the next most specific context, that has to be determined, does not depend on whether one has actually come from the first or second path. If we now require that all inherent paths of a message forwarding plan only intersect at most specific contexts, then the most specific contexts of the message forwarding plan can be determined statically from the most specific contexts of its inherent paths. So we have:

Let p be a message forwarding plan and L be the set of all inherent paths of p. Let all possible pairs of paths in L be non-interfering. Then a class which determines a most specific context in some inherent path of L determines a most specific context in p.

With this technique every object that has been retrieved by a forwarded message is colored by the objects which determine the most specific contexts the message has touched. For example, this will be the instances of the class ORDERING-CUSTOMER which represent Mr. Smith in the role of a customer of the ordered-products retrieved by the message ['Smith$^{\text{CUSTOMER-V}}$, OrderedProducts]. The objects in the most specific contexts model the necessary properties of the relationship between the object of interest and the retrieved object(s). If in a subsequent message the context color of an object is taken into account, methods of type (b) can be utilized in a dynamic view.

Consider the completed customer view of Figure 3. The class ORDERED-PRODUCT describes the behavior of context-colored product instances. A context-colored product instance is dependent on the particular customer instance for which the message forwarding plan which retrieved it has been executed; this is reflected in making ORDERED-PRODUCT a component-of CUSTOMER-V, and specifying its context ORDERING-CUSTOMER.

When the message forwarding plan b is executed for the customer instance 'Smith$^{\text{CUSTOMER-V}}$, all the product instances the customer Smith has ordered will be returned. Each instance will, however, be colored with the most specific contexts the message forwarding mechanism touches. E.g., in our case 'prod632'$^{\{\text{'Smith-ordering632'}\}}$ would be one such object returned, where 'Smith-ordering632' is the instance of ORDERING-CUSTOMER which represents Smith as customer of the product "632". In addition, each object returned will receive the color ORDERED-PRODUCT to change its behavior to that class. The subsequent message. ['prod632'$^{\{\text{'Smith-ordering632'}\}\text{ORDERED-PRODUCT}}$ OrderDate] will now be handled properly, as the coloring will appropriately restrict the scope of its execution; whereby the predefined method "context" is used to refer to the context of a oontext-colored object (see: Figure 3).

Using the same mechanism for the message forwarding plan to identify carriers that are able to ship the ordered products of a customer (see plan c in section 4) we finally arrive at the completed customer view CUSTOMER-V illustrated in Figure 3.

```
class CUSTOMER-V
  view of: CUSTOMER
  attributes:
    Credit: US$
  methods:
    ResponsibleSalesman: SALESMAN
      message forwarding plan
        ResidentIn REGION
        ResponsibleSalesman SALESMAN
    OrderedProducts: set-of
          ORDERED-PRODUCT-V
      message forwarding plan
        has-role ORDERING-CUSTOMER
        component-of PRODUCT
end CUSTOMER-V
```

```
class ORDERED-PRODUCT-V
  view-of: PRODUCT
  component-of: CUSTOMER-V
  context: ORDERING-CUSTOMER
  methods:
    OrderDate:     DATE
      message forwarding plan
      context ORDERING-CUSTOMER
OrderDate DATE
    PossibleCarrier: set-of
        POSSIBLE-CARRIER-V
      message forwarding plan:
        context ORDERING-CUSTOMER
        ((component-of PRODUCT
        constituent-of SHIPMENT-
        OFFER has-constituent
        CARRIER) intersect
        (role-of CUSTOMER ResidentIn
        REGION constituent-of
        SHIPMENT-OFFER has-consituent
        CARRIER))
end ORDERED-PRODUCT-V

class POSSIBLE-CARRIER-V
    ...
end POSSIBLE-CARRIERS
```

Figure 3: Completed view class CUSTOMER

6. Conclusion

The dynamic view definition technique introduced in this paper eliminates the need that for each user (group) all complete views, he/she will ever need, be predefined before any query can be executed. First of all, such work would place a very heavy burden on the DBA and the domain expert, but it would also lead to many problems and considerable amounts of work when the conceptual schema of the database has to be changed.

The method described is derived from methods developed for the dynamic creation of global views in multi-databases, and it is based on the idea that a user will always have some object(s) of interest in mind when he wants to work with the database. The view is then dynamically created around those object(s) of interest and can be behaviourially and structurally quite different from the conceptual (object oriented) schema. The paper introduces two concepts, message forwarding and object context coloring, to support (a), the construction of dynamic (object oriented) user views, and (b), the identification and control of the correct execution of complex queries even when large structural differences between views and conceptual level exist.

The techniques described are currently in the process of being implemented on Sun workstations. The prototype should allow the construction of user oriented dynamic views for multi-media databases, documents, materials, descriptions, designs, business data, etc. The dynamic view construction features allow a user to start useful work with a database, even when he/she has only very limited knowledge about the database technology employed and the properties of the data and data structures contained in the database on hand.

7. Acknowledgement

We are grateful to Markus Stumptner for his helpful comments on the paper.

8. Literature:

[BAN87] Banerjee J., et al.: Data model issues for object-oriented applications. In: ACM Transactions on Office Information Systems, Vol. 5, No. 1, 1987, pp. 3-26.

[BH86] Bryce, D. and R. Hull: "SNAP: A Graphics-Based Schema Manager." In: Proceedins of the Second International Conference on Data Engineering, Los Angeles, 1986, pp. 151-165.

[BM86] Brodie M., J. Mylopoulos (eds.): On Knowledge Base Management Systems. Springer, New York, 1986

[COD70] Codd, E.F.: "A Relational Model for Large Shared Data Banks." In: Comm of ACM, Vol 13, No. 6, 1970.

[COX86] Cox, B.: "Object-Oriented Programming: An evolutionary Approach." Eddison Wesley, Reading Massachussets, 1986.

[DS86] Dayal U. and J. M. Smith: "PROBE: A Knowledge-Oriented Database Management System." In: [BM86]

[DU87] Urban, S. D. and L. M. Delcambre: "Perspectives of a Semantic Schema." In: Proceedings of the Third International Conference on Data Engineering, 1987, pp. 485-492.

[FN85] Furtado, L. and E. J.. Neuhold: "Formal Techniques for Data Base Design." Springer, Berlin 1985.

[GK85] Goldman K.J., P.C. Kanellaiks, S.A. Goldman, S.B. Zdonik: "ISIS: Interface for a Semantic Information System". ACM SIGMOD 1983, pp. 328-342.

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

[HM81] Hammer, M. and D. McLeod: "Database Description with SDM: A Semantic Database Model." In: ACM Transactions on Database Systems, Vol. 6, No. 3, 1981, pp. 351-381.

[KM84] King R. and S.Melville: "Ski: A Semantics Knowledgeable Interface". VLDB, 1984, pp. 30-33.

[KNS88] Klas W., E. J. Neuhold and M. Schrefl: "On an object oriented data model for a knowledge base." In: Proceedings of the European Teleinformatics conference, North Holland, 1988.

[LAR86] Larson, J. A.: "A Visual Approach to Browsing in a Database Environment," IEEE Computer, Vol. 19, No. 6, June 1986, pp. 62-71

[LIT85] Litwin, W.: "Implicit joins in the multidatabse system MRDSM". IEEE-COMPSAC, 1985, pp. 495-504.

[MAI83] Maier, D.: The Theory of Relational Databases. Computer Science Press, 1983

[MBW80] Mylopoulos J., P.A. Bernstein, H.K.T. Wong: "A Language Facility for Designing Database-Intensive Applications." In: ACM Transaction on Database Systems, Vol. 5, No. 2, 1980, pp. 185-207.

[MOO86] Moon, D.: "Object-Oriented Programming with Flavors." In: Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications. Portland, Oregon, Sept. 1986.

[MOT86] Motro, A.: "Constructing Queries from Tokens". ACM SIGMOD 1986, pp. 120-131

[MU83] Maier, D. and J.D. Ullman: "Maximal objects and the semantic of universal relation databases." ACM Transactions on Database Systems, Vol. 8., No. 1, 1983, pp. 1-14

[MRSS87] Maier D., D. Rozenshtein, S. Salveter, J. Stein, D. Warren: "PIQUE: A relational query language without relations." In: Information Systems, Vol. 12, No. 3, pp. 317-335, 1987.

[NEU86] Neuhold, E.J.: "Objects and abstract data types in information systems." In: Proc. of the IFIP TC2 Working Conference on Database Semantics; R. Meersman, Steel T.B. (eds.). North Holland, 1986, pp. 1-12.

[NS88] Neuhold, E.J. and M. Schrefl: "Towards databases for knowledge representation." In: Schmidt J.C., Thanos C. (eds.): On Knowledge Based Management Systems II. Springer, Heidelberg, New York, 1988. (in print)

[SHI81] Shipman, D. W.: "The Functional Data Model and the Data Language DAPLEX." In: ACM Transactions on Database Systems, Vol. 6, No. 1, 1981, pp. 140-173.

[SN88a] Schrefl, M. and E. J. Neuhold: "Object Class Definition by Generalisation using Upward Inheritance". Proceedings of the 4th International Conference on Data Engineering, IEEE, 1988, pp. 4-13

[SN88b] Schrefl, M. and E. J. Neuhold: "A knowledge-based approach to overcome structural differences in object-oriented database integration". Proceedings of the IFIP Working Conference "The Role of Artificial Intelligence in Databases and Information Systems." Guangzhou, North Holland, 1988.

[SS77] Smith, J.M. and D.C.P. Smith: "Database Abstraction: Aggregation and Generalization." ACM Transactions on Database Systems, Vol. 2, No. 2, 1977, pp. 105-133.

Appendix

Definitions

Def. (message forwarding path):
(a) Let r be the name of a relationship and C the name of a class, then "r C" is a message forwarding path.
Note: Here the term relationship is meant to refer to relationships, attributes, and parameterless methods.
(b) If l is a message forwarding path, r is the name of a relationship and C the name of a class, then "l r C" is a message forwarding path.

Def. (message forwarding plan):
(a) Every message forwarding path is a message forwarding plan.
(b) If p and q are message forwarding plans, then the concatenation of p and q, pq, is a message forwarding plan.
(c) If p is a message forwarding plan, then the iteration of p, p^*, is a message forwarding plan.
(d) If p and q are message forwarding plans, then (p union q) and (p intersect q) are message forwarding plans.

Def. (result class of a message forwarding plan):
The result class of a message forwarding plan p, result-class(p), is defined as
(a) C, iff p="r C"
(b) result-class(q2), iff p=q1q2
(c) result-class(q), iff $p=q^*$
(d) result-class(q1)=result-class(q2), iff p=(q1 union q2) or p=(q1 intersection q2)

Def. (validity of a message forwarding plan):
A message forwarding plan p is valid for a class O, iff
(a) p="r C", and the relationship r is defined between the class O and the class C
(b) p=q1q2, and q1 is valid for O, and q2 is valid for result-class(q1)
(c) $p=q^*$, and q is valid for O and for result-class(q)
(d) p=(q1 union q2) or p=(q1 intersection q2), q1 and q2 are valid for O, and result-class(q1)=result-class(q2)

Def. (result class of a message forwarding plan):
The result class of a message forwarding plan p, result-class(p), is defined as
(a) C, iff p="r C"
(b) result-class(q2), iff p=q1q2
(c) result-class(q), iff $p=q^*$

(d) result-class(q1)=result-class(q2), iff p=(q1 union q2) or p=(q1 intersection q2), q1 and q2 are valid for O, and result-class(q1)=result-class(q2)

Def. (inherent path):
A message forwarding path l is an inherent path of the message forwarding plan p iff l can be obtained from p by a sequence of the following substitutions:
(a) if p has a subplan q^* then substitute q^* in p by q
(b) if p has a subplan q=(q1 union q2) or q=(q1 intersection q2), then substitute q in p by either q1 or q2.

Def. (predecessor):
Let $C_1,..C_n$ be the sequence of classes that appear in the given order in a message forwarding path l valid for O . Then the predecessor of a class C_i in l, predecessor(C_i,l), is defined
(a) as O, if i=1
(b) as C_{i-1}, if i>1, i<=n.

Def. (execution of a message forwarding plan):
The execution of p on an object o is defined as follows:
(a) if p = "r C", then the answer to the message "r:C" sent to o, i.e. [o r:C],is returned as result of p
(b) if p =q1q2, then q1 is executed on o, q2 is executed on the objects returned by the execution of q1, and the union of the results of these executions of q2 is returned as result of p.
(c) if $p=q^*$, then q is executed on o. If the result is the null object, then o is returned. Otherwise p is executed on all objects returned, and the union of the results of these executions is returned as result of p.
(d) if p=(q1 union q2) or p=(q1 intersection q2), then q1 and q2 are executed separately for o and the union (rsp. intersection) of the results of both executions is returned as result of p.

Def. (current context):
The current context of a message at the class O in a message forwarding path l valid for S, current-context(O,l) is:
(a) {}, iff l does not contain O
(b) context(S), if O=S
(c) current-context(predecessor(O),l), if current-context(predecessor(O),l) > context(O)
(d) context(O), otherwise

Def. (most specific context):
Let $C_1,..,C_n$ (n>=2) be the sequence of classes in a message forwarding path l valid for C_1. Then the class C_i determines a most specific context in l, iff
(a) i=1, and context(C_1) > context(C_2)
(b) i>1, i<n, and context(C_{i-1}) < context(C_i) > context(C_{i+1})
(c) i=n, n>1, and context(C_{n-1}) < context(C_n)

Def. (non interfering inherent paths):
We call two inherent paths l and k of a message forwarding path non-interfering iff
(a) every class C which determines a most specific context in l and which is contained in k determines also a most specific context in k, and
(b) every class C which determines a most specific context in k and which is contained in l determines also a most specific context in l.

Message forwarding rules for an inquiry message M received by the class O from the class C:

The inquiry message M is considered an object with the attribute OriginalMessage, and the components PreviousPath, AcquiredPaths and EncounteredCycles. The component PreviousPath is initialized with the object class of the addressee of the original message.

Rule 1: (Terminate the search on a success)
If the class O has a method which implements the response to the OriginalMessage, then the class O, the method selector and the result class of the found method are concatenated to the PreviousPath (from which the first class has been dropped) and recorded with the attribute AcquiredPaths in the answer to the inquiry message given to C.

Rule 2: (Terminate the search, if a cycle is encountered)
If the class O appears already in the PreviousPath and the first relationship in the encountered cycle is a has-category-relationship then mark O as start/end point of the cycle and record it with the attribute EncounteredCycles in the answer given to C.

Rule 3: (Do not reenter a previously left context)
If the class O is contained in the context of an object class T that appears in the PreviousPath and it is not contained in the contexts of all those classes that appear after T in the PreviousPath, then terminate the forwarding of the inquiry message.

Rule 4: (Avoid too many context switches)
If the number of context switches is greater than ContextSwitchLimit, then terminate the forwarding of the inquiry message.

Rule 5: (Forward the inquiry message to all related classes)
If no other rule applies, forward M to all classes related to O, except to C, . From the answers given, take the union of the encountered cycles and the union of the acquired paths to determine the value of the attributes EncounteredCycles and AcquiredPaths for the answer to C.

Plan combination rules

P_C denotes the equivalence class of plans which intersect for the first time at the object class C after a possible common prefix.

Rule 1: (Favor plans which preserve the context)
If in every inherent path of some plan no context switch occurs, then discard those plans in which in some inherent path a context switch occurs.

Rule 2: (Favor plans which preserve the context at least locally)
If some plan p=rst of some equivalence class P_C, where result-class(r)=R is the last object class in the common prefix of the plans in P_C and result-class(rs)=C, exists such that C is reached from R in every inherent path without any context switch, then discard those plans in PC in which in some inherent path a context switch occurs in the subpath from R to C.

Rule 3: (Avoid unintended context switches)
If in some plan p=rst of some equivalence class P_C, where result-class(r)=R is the last class in the common prefix of P_C and result-class(rs)=C, some unapproved context switch occurs in the path from R to C, then let the context switch be approved by the user.

Rule 4: (Take the intersection of two subplans which come from different context generalizations and meet with the same current context)
If two plans p=rst and q=rvw, where result-class(r)=R is the last class in the common prefix of P_C and result-class(rs)=result-class(rv)=C, exist in some P_C such that
(a) current-context(C,p) = current-context(C,q), and
(b) current-context(predecessor(C,l),p) < context(C,q) for every inherent path l in p which contains C, and
(c) current-context(predecessor(C,l),q) < context(C,q) for every inherent path l in q which contains C,
then replace p by p'=(r(s intersect v)t) and q by q'=(r(s intersect v)w) in P_C.

Rule 5: (Take the union of two subplans which meet with different current contexts)
If two plans p=rst and q=rvw, where result-class(r)=R is the last class in the common prefix of P_C and result-class(rs)=result-class(ru)=C, exist in some P_C such that for every inherent path l in p and for every inherent path k in pj: current-context(C,l) is not contained in current-context(C,k) and vice versa, then replace p by p'=(r(p union q)t) and q by q'=(r(p union q)w)

Rule 6: (Let the user himself decide in ambiguous situations)
If two plans p=rst and q=rvw exist in some P_C, where result-class(r)=R is the last class in the common prefix of P_C and result-class(rs)=result-class(rv)=C, then ask the user either to provide a set-operator, <set-op>, in order to replace p by p'=(r(s <setop> v)t) and q by q'=(r(s <setop> v)w) or to decide to discard p or q at all.

Rule 7: (Resolve circular dependencies among equivalence classes)
If the equivalence classes P_{Ci1},..,P_{Cin} are circularly dependent and the plans in every P_{Cij} (j=1..n) have already been combined, then look for the plan p=ruv of PC_{ij} (j=1,..,n) where result-class(r)=C_{ij} is the last class in the common prefix of P_{Cij} and result-class(u)=C_{ik} (k=1,..n) such that the length of the path u is maximal. Let p be this plan. Then replace p by p'=wv where w is the common prefix till the class C_{ik} of the plans in PC_{ik}.

Rule 8: (Add cycles to the initial plan)
If P contains only one equivalence class with only one message forwarding plan p, then continue with the plan&cycle combination rules.

Plan & cycle combination rules

Rule 1: (Neglect dangling cycles)
If the start-class S of some acquired cycle k^* does not appear in p, then discard k^*.

Rule 2: (Neglect cycles which interfere with the remaining plan)
If some acquired cycle k^* intersects with p at another class in addition to the start-class S of k^*, then discard k^*.

Rule 3: (Neglect cycles over different contexts)
If k^* is a cycle with start/end-class S and a context switch occurs in the path Sk, then discard k^*.

Rule 4: (Avoid circling the same context)
If k^* is a cycle with start/end-class S and no context switch occurs in the path Skk, then discard k^*.

Rule 5: (Combine cycles with the same start-class)
If the cycles k_i^* and k_j^* have the same start-class S and S appears in p, then ask the user to combine k_i^* and k_j^* properly.

Rule 6: (Accept cycles which may terminate)
If the start-class S of the cycle k^* appears in p such that S is followed by a has-category-specialization relationship to a different class than the first has-category-specialization relationship in k^*, then insert k^* after S in p.

Rule 7: (Have the developed plan confirmed by the user)
If no other rule applies, then have p confirmed by the user as final message forwarding plan.