Asynchronous Data Retrieval from an Object-Oriented Database

Jonathan P. Gilbert
Lubomir Bic
Department of Information and Computer Science,
University of California, Irvine, CA 92717, USA.

Abstract

We present an object-oriented semantic database model which, similar to other object-oriented systems, combines the virtues of four concepts: the functional data model, a property inheritance hierarchy, abstract data types and message-driven computation. The main emphasis is on the last of these four concepts. We describe generic procedures that permit queries to be processed in a purely message-driven manner. A database is represented as a network of nodes and directed arcs, in which each node is a logical processing element, capable of communicating with other nodes by exchanging messages. This eliminates the need for shared memory and for centralized control during query processing. Hence, the model is suitable for implementation on a multiprocessor computer architecture, consisting of large numbers of loosely coupled processing elements.

1. Introduction

The overall goal of the semantic data modeling project at UCI is to develop a semantic database system suitable for highly parallel processing. We believe that this can be accomplished if the underlying model is completely message-driven, i.e., without any centralized control and centralized memory. First, however, the semantics of the model and its operations must be defined. Based on these definitions, procedures that govern the propagation of messages during processing can be derived.

The present paper is a first step toward such a model. It describes the basic philosophy of our approach, the components of the model, and the semantics of queries. We also outline the *generic* procedures that permit queries to be executed in a purely message-driven manner.

The model has all the desirable features of a conceptual modeling system. These features are well known and have been presented many times before: see, for example, [BRODIE80, BORGIDA87]. In particular, the model combines the virtues of four concepts: the functional data model

[SHIPMAN81], a property inheritance hierarchy (common to most semantics networks [FINDLER79] and some frame based languages like KRL [BOBROW77]), the principles of message-driven computation [ARVIND78, AGHA85], and the data hiding/abstract data types of object-oriented programming systems [STEFIK86].

The paper is organized as follows: In section 2, we describe the representation and organization of base and derived data within our paradigm. We also sketch the syntax of queries and specify their semantics. Section 3 shows how requests can be processed asynchronously by propagating messages through the database hierarchy. Finally, section 4 contains some concluding remarks and points out the relationship of our model to some other approaches.

2. Components of an Object-Oriented Model

In this section, we begin by describing the representation, components and organization of data in our model. After the basics have been described, we present the message passing strategy.

2.1. Data Representation

A database is represented by a network of nodes and directed edges. Each node represents an independent database object. We adopt the philosophy found in many semantic data models (see, for example, [CODD79, HAMMER81, BANERJEE87]): higher-level (molecular) objects are recursively constructed from simpler database objects. Nodes of the network represent objects within the database enterprise (for example, people, colors, automobiles, or engines) and arcs represent various associations among these objects. There are two basic kinds of association: the IS-A relationship and the ROLE relationship. The first is used to construct an *inheritance* hierarchy (see, for example, [DAYAL84]) while the second is the functional "glue" that binds together molecular structures. These associations and the overall structure of a database is similar to those in an Omega knowledge base [ATTARDI86]. Data are organized in an *incremental* fashion, with more refined data descriptions beneath their more general ancestors' descriptions in the IS-A hierarchy. Figure 1 shows a single branch of a "modes-of-transportation" hierarchy. It is used to illustrate various aspects of the two hierarchies found in our model.

We distinguish two types of nodes: ellipses which represent sets of non-decomposable *atomic* objects and rectangular boxes which represent sets of compound *molecular* objects. The IS-A hierarchy (in which nodes are connected by the *unnamed* arcs) facilitates *inheritance* of properties and relationships, represented by ROLE associations. The arrows of the IS-A hierarchy show the direction in which inheritance takes place. We chose the name "role", rather than property, function, or relationship, to stress the fact that molecular objects are recursive compositions of simpler objects and each of the simpler objects plays a certain role in the "super" object. A database user may *choose* those roles he perceives as inherent (attributes) parts of an object and those which are more like relationships between independent objects. The former are displayed

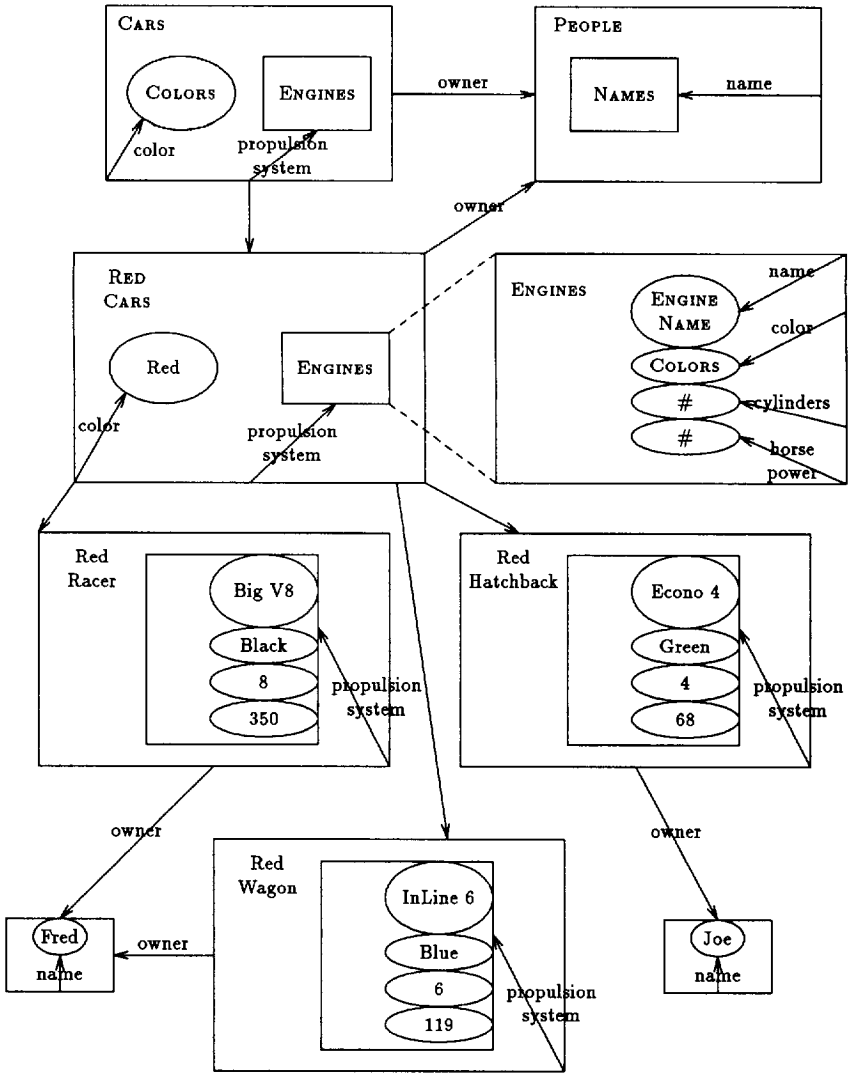

Figure 1
Single Branch "Modes-of-Transportation"

inside the objects description while the latter are displayed outside of the object. For example, in figure 1 the roles *color* and *engine* are perceived as part of an automobile while the role *owner* is identified as a relationship between an automobile and a person. Notice that these choices and many other choices related to the users' perception of the data are subjective. Although meaningful to the user, whether a role is displayed inside or outside of an object is irrelevant to the semantics of the database itself. On the other hand, there may be some roles (displayed inside or outside the node) which are absolutely essential to the description of an object. We call this type of role a *key role*; all other roles are *ordinary roles* (which may or may not be instantiated in all leaves).

For example, in figure 1 it the owner role (from cars to people) is key because (in this very simple world) all cars must be owned by people. However, if we were to look at that same relationship in the other direction we would find that it is *not* key because some, not all people own cars.

There is no explicit distinction made between sets of objects and individual elements in our model. Conceptually, each node contains a *generic description* of an object so that leaf nodes of the IS-A hierarchy are sets containing exactly *one* object. However, since there are relatively few *internal* (non-leaf) nodes it is desirable to store the bulk of the description and semantics at this level thereby minimizing the amount of redundant information at the "element" or *leaf level.* Furthermore, internal nodes serve a dual purpose: they represent the *set* of leaves reachable by following outgoing IS-A arcs and they serve as a *type* for those leaves. One major advantage of this uniform view of sets and elements can be illustrated by the following simple example: If we assume that CARS is a multi-set, then, if Fred owned a fleet of identical Red_Racers, instead of just one, it would not be necessary to repeat the Red_Racer's description for each car. Conceptually, the current Red_Racer node would become a generic description and *empty* children nodes would be inserted to represent the individual automobiles.

2.2. The External Schema

The global external schema contains only non-leaf nodes (set description objects). It describes for the user the entire database enterprise in a single connected graph. Even though leaf nodes (object instances) are not included in the global schema, the schema is often too large to display as a single graph; therefore, the user may view the global schema as several graphs rather than a single graph. The system provides an interactive graphics browser that permits users to explore the schema. An object is selected as the current point of interest. This node and the nodes which are *directly* connected to it by a single IS-A or *role* arc are displayed in a window for the user. For example, when displaying the PEOPLE object's node in figure 1 nodes representing CARS and NAMES would also be shown (without any further detail). The user can navigate through the schema (change the point of interest) by moving a mouse pointer to an object and pressing the appropriate mouse button. The new node's object then becomes the point of interest. Many objects and arcs in a schema are not *base* but *derived* (shown as dashed boxes and arrows). Base objects have a concrete representation *stored* in the database while derived (or virtual) data (described in more detail later) are calculated by applying rules when a user tries to "retrieve" that data. In the day to day interactions with the database, there is no visible difference between virtual and stored data for the user except that virtual data cannot be directly updated.

2.3. Derived Data

Much of the semantic richness of this model comes from its support of a variety of derived data. There are two types of derived data: sets and roles, which are represented by rules that are

part of an object description. The syntax of these rules is beyond the scope of this short paper but we do discuss the derived data available and, in the next section, the data retrieval algorithms including the instructions necessary for retrieving data from virtual objects and arcs. To better illustrate the three kinds of *union-subset* and *aggregate data*, we present a non-trivial example (shown in figure 2) which is based on examples in [McLeod78]. Note that dashed nodes and arcs represent derived data.

2.3.1. Derived Sets

Union-subset nodes are a grouping mechanism which allow the formation of heterogeneous sets. All union-subset nodes contain pointers to the base sets that are the basis for a set abstraction. There are three types of union-subset abstraction called *category*, *collection* and *power sets*. To define a derived set a user must specify: its name, its type, the sets whose union are the basis for the (maximal) derived set, restrictions on each set's roles (if any), and any new roles which are associated with objects in the virtual set.

Collection sets "automatically" include all leaves in all base sets which are in the union and whose descriptions are consistent with any restrictions placed on that set's roles. In figure 2 oil tankers is a collection because its members are all military and merchant ships whose class is "oil tanker". Unlike collection sets, a *category set's* node contains *explicit* pointers to its members which have been specifically inserted into that category. Banned Ships (see figure 2) are an example of a category. There is no rule associated with the banned ships object. Any ship may be banned but a user must explicitly ban it. *Power sets* can be thought of as a generalization of the category. The major difference between them is that the power set is based on the *power set of the union* of some base sets instead of their union — each *element* of a power set is a *category*. In figure 2 convoys are modeled as a power set because each convoy is a set of ships and not a single ship. Notice that the roles (location and max-speed) are associated with the convoy and not the individual ships in that convoy.

2.3.2. Derived Roles

Virtual role abstractions are classified by the action taken by the system when it instantiates them. Actions correspond to substituting a subquery for the virtual role, spawning the new query which is reprocessed by the node and "creating" a virtual arc or a virtual node. A *VR-arc* rule causes a virtual arc to be "created" while a *VR-node* causes a virtual node to be "created".

To create a VR rule a user must specify: the name of the role, the set on which it is defined, the domain of the operation (where the rule is mapped to) and the operation itself (which may be anything from a simple "restriction list" to a general purpose (external) procedure or both). In addition, the user must determine whether the rule will be evaluated at the set or instance level of the IS-A lattice.

An example of a VR-node abstraction is *aggregate data.* Aggregate data are defined by aggregate operators which abstract a single object from a set of objects. Examples of aggregate operations are: calculating the maximum speed of a convoy (see figure 2) or determining the average length of an oil tanker (not shown in the figure).

VR-arc abstractions are *inference rules*, so called because the relationship which they make explicit can be inferred from the structure of schema anyway. Information is retrieved by substituting a role request subquery for a VR-arc "role" thereby "creating" the virtual arc. For example, consider the *grandfather* relationship between people. This could be represented explicitly as a role (arc) from an individual to his parents' fathers or it could be represented *implicitly* by including a rule which states: "To find a person's grandfather, first find his parents and then find their fathers."

To the user, derived data of both kinds can be used to retrieve information in exactly the same way as any base role.

3. Message-Driven Processing

In an object-oriented environment, each object is an *abstract data type* which includes a description of the data it represents and a set of operations (*methods*) for manipulating that data. These methods are triggered when messages are received from other objects. The data representation is not visible to the outside world; the user "sees" a "black box" and the actions (which may vary from one abstraction to another) for the manipulation data inside the box. In our model, a similar situation exists except that communication between objects is achieved by a small number of *generic* methods.

The object-oriented paradigm with its abstract data types and message passing semantics make our model suitable for implementation on a highly parallel loosely coupled multiprocessor. The ideal architecture has no centralized control or memory and each node may be mapped onto a different processing element (PE) as long as there are physical communication paths for each logical arc. There are many architectures that satisfy this requirement.

3.1. Internal Representation of Arcs and Objects

Objects are data structures that are mapped onto the local memory of a processor (PE). The description of an object contains information about all data within that object. It must include components that represent arcs, derived data and operations (or *methods*) that are triggered by incoming messages. In addition, the description contains information about individual roles: i.e. which of them are *key* and where they are to be displayed. We have shown that roles' nodes may be *displayed* inside and outside of their "super" object's node. The semantics of these differences are in some sense "external". This means that, although the placement of a role node may make a difference to the way in which a user perceives a concept, placement makes no difference to the

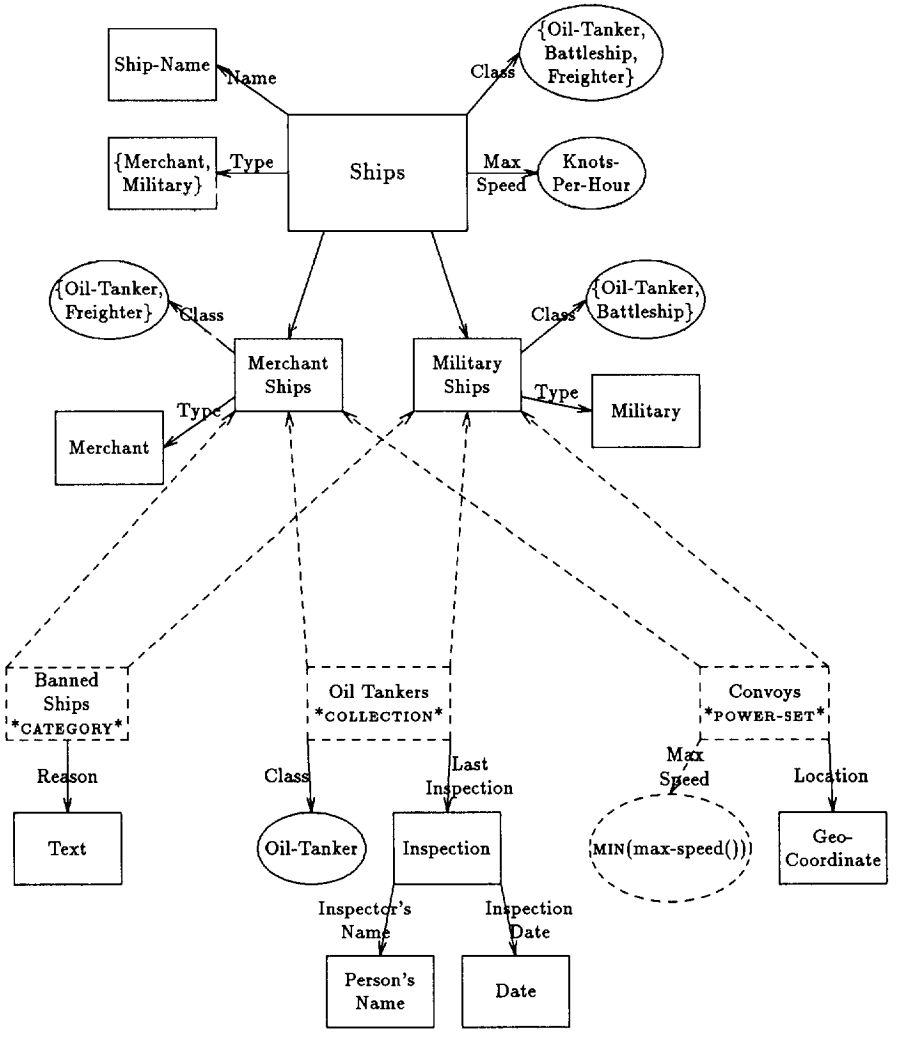

Figure 2
An Example of Derived Data

way that the system processes a query on an object. On the other hand, the difference between key and non-key roles are internal since they are absolutely essential to the description of an object.

Arcs represent either IS-A or ROLE relationships between objects; they are implemented by using pointers where each pointer identifies a PE and an address within that PE's local memory space. All arcs are bi-directional which means that each arc is actually represented by *two* pointers, one at each of its ends. Atomic roles are not represented by independent objects. Since atomic objects are simple values, it would be wasteful to have independent objects that just return a value. Instead, we store *singleton* roles locally so that they can be retrieved from an object's local memory without flooding the system with unnecessary messages.

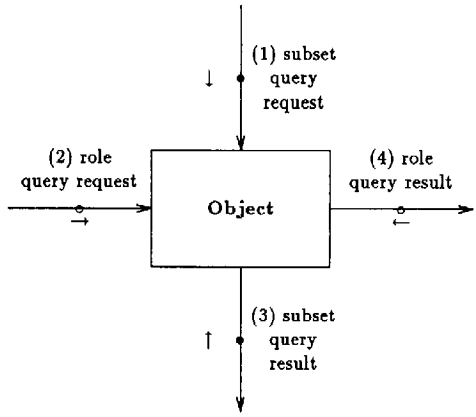

Figure 3
The Four Message Types

3.2. Information Retrieval

Queries are formulated and processed against the external schema. There are two kinds of information retrieval queries. The first variety of retrieval request refers to an object as a *set* while the second refers to it as a *type*. A user may want to retrieve all elements of a set which have particular properties (we call this kind of request a *subset query*) or a user may want information about the objects associated with a particular role. (This second type of query is called a *role query*.) The basic strategy is for the user to send a message to the injection point node which either replies to the request directly or propagates the query to other objects and waits for their response. When all objects have responded, the node can combine the results and return the result to the sender. This query processing strategy and the two query types are implemented using four types of generic message. These messages are called: (1) the subset query request message, (2) the role query request message, (3) the subset query result message and (4) the role query result message. The four message types are illustrated in figure 3. Note that the arcs at the top and bottom of the object represent IS-A relationships and arcs on the sides represent role relationships.

By examining the message, an object can determine which action it should take (there is exactly one action for each message type). We now describe the general strategy and show high-level descriptions of the procedures used to process user requests.

Conceptually, a request for information either points to a set of objects and retrieves the subset of those that satisfy some list of restrictions on their outgoing roles or retrieves information about some of an object's roles. Restrictions are recursively decomposed and applied to objects reached via role arcs starting at the original object, until the entire restriction is satisfied or fails. First we give the basic syntax of queries. Each query can be thought of as a four-tuple:

⟨⟨set⟩; ⟨query-type⟩; ⟨query-restriction⟩; ⟨query-output⟩⟩ where:

⟨set⟩ $\stackrel{\text{def}}{=}$ the name of the injection point node.

English "equivalents" of the queries are shown in *italics*; comments are shown in roman font.

1. *List all Red Cars Owned by a Person Named Fred*: The key word here is "list" the system produces a list of cars.
   ⟨RED CARS; SUBSET-REQUEST; owner.name = "Fred"; LIST(VALUE(ALL))⟩.

2. *Are there any Red Cars Owned by a Person Named Fred*: This time a "yes" or "no" answer will be produced.
   ⟨RED CARS; SUBSET-REQUEST; owner.name = "Fred"; EXISTS(ALL)⟩.

3. *Is it possible that a Person Named Fred could be the owner of a Red Car*: This is a query about the owner role and not the set of Red Cars.
   ⟨RED CARS; ROLE-REQUEST; owner.name = "Fred"; EXISTS(ALL)⟩.

Figure 4
Sample Queries

⟨query-type⟩ $\stackrel{\text{def}}{=}$ identifies the query as a role request or a subset request.

⟨query-restriction⟩ $\stackrel{\text{def}}{=}$ a set of paths which define the restrictions on roles involved in the query. Its format is comparable to the body of the *is-there?* query in Omega [ATTARDI86]. The processing, however, is not the same.

⟨query-output⟩ $\stackrel{\text{def}}{=}$ decribes roles and format of the output of the query.

To illustrate the expressive power of these queries and to provide a set of concrete examples for subsequent discussions, consider the list of queries in figure 4.

When processing any query, the system must differentiate between *key* and *non-key* roles. The reason for this is obvious: If a role is key to a set's object then it definitely exists for *all* instances of that set; if it is non-key then it may exist in some of a set's instances. Notice that this definition of *key* is quite different from a key attribute in many traditional database models since uniqueness is not necessary.

There are two kinds of question that can be asked about a role: (1) does the role exist and (2) if it exists, does it map to a particular set of objects or values. The semantics of a role request query are captured by the two procedures shown in figure 5. A query names an injection point r and lists the roles (and restrictions on those roles) which are the focus of the query. A status value is calculated for all roles named in the query by sending a role request (sub)query message along each of the named arcs. Each role object processes it's subquery independently of all other role objects and the strategy is exactly the same as that followed at the injection point. The overall strategy is that the query is *dynamicly* recursively decomposed for parallel processing. Eventually, for each role path a *terminal* node is reached. A terminal node is a node which can determine a status (and a value) for a particular (sub)role; it is *not* necessarily a leaf node. Once the status is

```
Procedure Role-Query-Request (Triggered by a message of type 2)
  create activity record for pending query &
  for each path R in query-restriction
    if head(R) is a base role then
      if it is a singleton or node is terminal
        then send a Role-Query-Result message to self
        otherwise remove R from the restriction list &
          send a Role-Query-Request message
            containing tail(R) along arcs that match head(R)
      otherwise (R is a virtual role)
        if node is not a leaf and R is a "set-level" rule
          OR if node is a leaf and R is a "instance-level" rule
            spawn appropriate subquery


Procedure Role-Query-Result (Triggered by a message of type 4)
  store result
  if last result for corresponding activity
    determine status of query
    Case 1: the original query was a Role-Query-Request
      SubCase 1.1: the object is a base set
        send a Role-Query-Result message to sender & destroy activity record
      SubCase 1.2: the object is union-subset node
        & its base sets have not been visited
        & the query has NOT definitely succeeded or failed
          for each base set in union
            send a Role-Query-Request message containing only unfound roles
          adjust activity record to reflect change in query
      SubCase 1.3: the object is union-subset node
        & its base sets have not been visited
        & the query has definitely succeeded or failed
          send a Role-Query-Result to sender & destroy activity record
      SubCase 1.4: the object is union-subset node
        & the result comes from a base set
          store result & destroy activity record
          if it also is the last result for original activity record
            then determine status of original query (minimum status found) &
              send Role-Query-Result to sender & destroy activity record
    Case 2: the original query was a Subset-Query-Request
      if node is not a leaf & status is not 5
        then for each non-leaf child send Subset-Query-Request message to child
          if status is 1, 2 or 3
            then for each leaf child send Subset-Query-Request message to child
          adjust original activity record to reflect change in query
        otherwise (the node is a leaf)
          send a Subset-Query-Result to sender
          destroy activity record
```

Figure 5

Role Request Procedures

known it is returned (on a role query *result* message) along the arc on which the original request arrived. When a non-terminal node has collected results from all its subqueries, they are used to

determine its own status which is then sent back to the sender of the request. Note that because of the distributed structure of the database and the absence of centralized control in this strategy, the subqueries are distributed and the results collected in an asynchronous manner.

There are five possible status values for individual roles; their most general meanings are listed below. Note that, although all five status values are not necessary for processing role request queries, they are all necessary when processing subset requests.

1. This role was found and (the restrictions on it) satisfied for all possible instances of the set rooted at this node (for key roles only).
2. This role definitely exists for all possible instances, however, the restriction on this role may not be satisfied (once again key roles only).
3. This role was found and exists for some instances of the rooted set (for non-key roles only).
4. This role was not found.
5. This role was found and is definitely not satisfiable for any instance of the rooted set.

The *maximum* value of the individual roles' status values is taken as the status of the query for the entire object. The basic meanings of the object status values (used by all query types) are listed below:

1. All restrictions (on roles) were satisfied.
2. All restricted roles definitely exist but some *may* not be satisfied.
3. Some restricted roles may exist for some instances and not others.
4. Some resticted roles were not found.
5. Some restricted roles are definitely not satisfiable.

The semantics of subset query request processing is slightly more complicated because subset queries spawn role queries. Figure 6 shows sketches of the two procedures executed by a database object when it receives a subset query message. The processing strategy depends on the propagation of messages from the injection point down through the IS-A hierarchy possibly all the way to the leaves. At each node visited, subset query requests spawn role request subqueries to determine whether individual restrictions have been satisfied. There are four basic assumptions about what happens to object descriptions as the IS-A hierarchy is traversed towards the leaves: (1) more role descriptions may be added, (2) *any* role's definition may become more restricted, (3) *non-key* roles may become *key* or so restricted that they "disappear" and (4) *virtual* roles are treated like *non-key* roles.

The semantics of a subset request query are captured by the two recursive procedures shown in figure 6. They are applied as follows: the query names a node s as the target set, from which elements are to be retrieved; S represents the set of nodes reachable from s by following IS-A arcs and L is a subset of S containing only leaf nodes (elements). Each element of L is an object which may be retrieved by the query, if it satisfies the specified restrictions.

In each element of S, the status of all roles named in the query is determined by sending role request queries along all role arcs listed on the query restriction list. In each node of the set S–L (i.e., non-leaf nodes), a status is determined for each role by the role request query which is compared with the status obtained by the node's parent. This is necessary because some non-key

```
Procedure Subset-Query-Request (Triggered by a message of type 1)
  if the node is a leaf & query originated from a category node &
    object is not directly connected to that category node
      then (report failure) send a Subset-Query-Result to sender
      otherwise create activity record for pending query
        for each path R in query-restriction
          if head(R) is a base role then
            if it is a singleton or node is terminal
              then send a Role-Query-Result message to self
              otherwise remove R from the restriction list &
                send a Role-Query-Request message
                  containing tail(R) along arcs that match head(R)
            otherwise (R is a virtual role)
              if node is not a leaf and R is a "set-level" rule
                OR if node is a leaf and R is a "instance-level" rule
                  spawn appropriate subquery


Procedure Subset-Query-Result (Triggered by a message of type 3)
  store result &
  if last result for corresponding activity
    then determine status of query & send Subset-Query-Result to sender &
      destroy activity record
```

Figure 6

Subset Request Procedures

roles "disappear"; if the previous status was **3** and the current status is **4** then the current status must be changed to **5**. The object's status is then calculated and if it is *not* **5** then the query (including the status values) is passed to its descendants. Nodes in the set L determine the status in a similar way. This final value determines whether the object satisfies the given query; if it does, the data specified in the query's output field are retrieved and output.

Notice that all non-singleton role status values are calculated independently and that an *object* must wait for all of its roles to report their status before it continues processing a query. The first observation suggests a potentially high degree of parallelism if the system is implemented on a loosely coupled multiprocessor architecture. The second observation seems to imply that any benefit from this parallelism is lost because objects spend much of their time waiting for results from other objects. This conclusion is incorrect for several reasons: First, the fact that objects spend much of their time waiting does *not* imply that PEs are *busy waiting* or even idle. When a *PE* receives a request message, it creates an activity record for the request and when all the necessary subqueries have been spawned, it stores the activity record until it receives result messages for that request. When a result message is received, the PE determines whether it is the last result for the query; if it is not, the message is stored with the activity record. Otherwise, it is combined with the other results in order to calculate the object's status. This strategy allows for true asynchronous processing of queries and enables a high degree of parallelism without using a database management system query optimizer.

3.2.1. An Example — Processing a Simple Query

To clarify our asynchronous query processing strategy, we will describe the processing of the first sample query shown in figure 4. In order to satisfy that request, it is propagated through the schema shown in figure 1. We assume that all roles are key and that, initially, the status of the query and all of its roles are **4** (not found). Since the user requested all information about red cars the system will add all RED_CARS' roles that are not explicitly mentioned in the query to the ⟨query-restriction⟩ (in this case there are just two: propulsion-system and color). Note that these new roles can be assigned a status of **1** and, therefore, do not add significantly to the processing time. When the RED_CARS object receives the subset request, it decomposes the ⟨query-restriction⟩, stores the status of propulsion-system and color, and sends a role request message to PEOPLE. The ⟨query-restriction⟩ of this new message contains *name = "Fred"* and since name is a singleton the PEOPLE object determines that "Fred" is a (not the) valid name and, therefore, returns a status of **2** to RED_CARS. The RED_CARS object then calculates the status of the query by taking the maximum of the roles' status values: **2**. From this status RED_CARS determines that any of its children may satisfy the query and it sends subset request messages to the Red_Racer, the Red_Hatchback and the Red_Wagon. If each of these objects is mapped to a different PE then each will be able to look up its singleton roles and send role request messages to its non-singleton roles independently and in parallel with the other objects. Eventually, Red_Hatchback determines that its status is **5** and, therefore, it returns its status but no data. At the same time Red_Racer and Red_Wagon determine that their status values are **1** and they, therefore, do return data. When RED_CARS has received subset results from each of its children it combines the successful results and returns the objects' descriptions to the user.

4. Conclusions

Similar to other semantic and object-oriented database models, our approach has a clear advantage over the classical database models. The classical models are relatively low-level and capture little of the semantics of the application domain.

There have been many research efforts directed towards improving the semantics of database modeling and several surveys have been published on the subject — see, for example, [BIC86, HULL86]. Some research has produced significant enhancements to the relational model. For example, J. Smith and D. Smith added aggregation and generalization abstractions to the relational model (both of which are integral parts of our model) to produce their hierarchical semantic model [SMITH77]. Codd also introduced an enhancement to the relational model [CODD79] (known as the Tasmania relational model) which includes many forms of abstraction (including aggregation and generalization). Another approach has been to develop new semantic models which replace the relational data model; Hammer and McLeod's SDM [HAMMER81] is a good example of this. A major drawback of both the latter models is their *extreme* complexity — only the most sophisticated

users may find them useful modeling tools. By comparison, object-oriented models like ORION [BANERJEE87] and this model are very simple to use.

We believe that object-oriented models have some advantages over each of the semantic models. In particular, in object-oriented approaches, objects include the procedures (methods) for manipulating the data which they contain. Because objects communicate by sending each other messages and their methods are independent local procedures, there is an excellent *potential* for parallel processing. Finally, because of the *generic* methods which are built into its objects, our model provides a general framework for the development of database applications.

REFERENCES

[AGHA85] AGHA. G.A. Actors: A Model of Concurrent Computation In Distributed Systems. Tech. Rep. No. 844. MIT Artificial Intelligence Lab., MIT, Cambridge, Mass..

[ARVIND78] ARVIND, GOSTELOW, K.P. AND PLOUFFE, W. An Asynchronous Programming and Computing Machine. Tech. Rep. No. 114a. Univ. of CA., Irvine, Dept. of Info. and Comp. Sci..

[ATTARDI86] ATTARDI, G. AND SIMI M. A Description-Oriented Logic for Building Knowledge Bases. *Proc. of the IEEE 74*, 10 (Oct., 1986), 1335–1344.

[BANERJEE87] BANERJEE, J. ET AL. Data Model Issues for Object-Oriented Applications. *ACM Trans. on Office Information Systems 5*, 1 (Jan., 1987), 3–26.

[BIC86] BIC L. AND GILBERT J.P. Learning from AI: New Trends in Database Technology. *Computer 19*, 3 (Mar., 1986), 44–54.

[BOBROW77] BOBROW D.G. AND WINOGRAD T. An Overview of KRL. *Cognitive Science 1* (1977), 3–36.

[BORGIDA87] BORGIDA, A. Conceptual Modeling of Information Systems. In *On Knowledge Base Management Systems*, Brodie, M.L. and Mylopoulos, J., Ed., Springer-Verlag, 1987.

[BRODIE80] *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, Brodie, M.L. and Zilles, S.N, Ed., Sponsored by the Nat'l. Bureau of Standards, ACM SIGART, SIGMOD and SIGPLAN, Pingree Park, Colordo, 1980.

[CODD79] CODD, E.F. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems 4*, 4 (Dec., 1979), 397–434.

[DAYAL84] DAYAL, U. AND HWANG, H.-Y. View Definition and Generalization for Database Integration in a Multibase System. *IEEE Trans. on Software Engineering SE-10*, 6 (Nov., 1984), 628–645.

[FINDLER79] *ASSOCIATIVE NETWORKS Representation and Use of Knowledge by Computers*, Findler, N., Ed., Academic Press, 1979.

[HAMMER81] HAMMER M. AND MCLEOD D.J. Database Description with SDM: A Semantic Data Model. *ACM Trans. on Database Systems 6*, 3 (Sept., 1981), 351–386.

[HULL86] HULL, R. AND KING R. Semantic Database Modeling: Survey, Applications, and research Issues. Tech. Rep. No. TR-86-201. U.S.C., Comp. Sci. Dept..

[MCLEOD78] MCLEOD, D. A Semantic Data Base Model and its Associated User Interface. Rep. No. MIT/LCS/TR-214. Lab. for Computer Sci., MIT, Cambridge.

[SHIPMAN81] SHIPMAN, D.W. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. on Database Systems 6*, 1 (Mar., 1981), 140–173.

[SMITH77] SMITH, J.M. AND SMITH D.C.P. Database Abstractions: Aggregation and Generalization. *ACM Trans. on Database Systems 2*, 2 (June, 1977), 105–133.

[STEFIK86] STEFIK, M. AND BOBROW D.G. Object-Oriented Programming: Themes and Variations. *The AI Magazine 6*, 4 (Jan., 1986), 40–62.