INHERITANCE IN COMPUTER-AIDED DESIGN DATABASES: SEMANTICS AND IMPLEMENTATION ISSUES

Ellis E. Chang and Randy H. Katz

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Object-oriented models are advocated as good ways to describe semantic-rich application domains such as computer-aided design. We describe the role and limitations of conventional type-instance inheritance for modeling CAD data, and propose a new semantics based on instance-to-instance inheritance. New algorithms for object clustering, buffer management, and on-line back-up are described that exploit the new inheritance semantics for more efficient run-time performance.

1. Introduction

Object-oriented databases have become a popular research topic as the interest in applying database technology has moved from conventional commercial environments into new application areas. Object concepts from the programming language community are being adopted in database systems to provide persistent object platforms for artificial intelligence and programming language environments [ZDON84, ATWO85, MAIE86, ROWE86, ROWE87].

Our work has concentrated on applying database techniques in the computer-aided design environment [KATZ87]. CAD data is not well matched to formatted records, nor do existing database models capture the rich interrelationships among CAD objects, such as configurations, versions, and correspondences across representations of the design. Object-oriented databases provide a more natural way to structure these semantic-rich applications.

For us, the key aspect of an object-oriented system is the mapping of an application's data manipulation logic into a set of *abstract data types*, with associated operations and attributes. In addition, these systems often support the concept of *type hierarchies*, i.e., a taxonomic classification is applied to instances, which belong to types, which in turn belong to supertypes, etc. Operation and attribute definitions can be propagated along the lattice formed by instances, types, and supertypes through *inheritance* mechanisms. Thus, operations and attributes defined within a type are callable and usable by all its subtypes and instances, providing a short-hand specification of definitions as well as information hiding.

Inheritance is a powerful mechanism, but as it is commonly implemented between types and instances, it is not well-matched to computer-aided design applications. These require information to be propagated among objects in more complex ways, such as between an ancestor and a descendent version. Thus, we must reanalyze inheritance semantics and its impact on the implementation of the components of a database system. In this paper, we first describe a base level Version Data Model used by our prototype object-oriented system for CAD data. In Section 3, we discuss conventional inheritance semantics and an alternative that is better suited for CAD applications. This *instance-to-instance* inheritance can be incorporated into the model of Section 2. New implementation algorithms for smart clustering and buffering, and tuned for the new inheritance model and the complex space of interrelationships, are described in Section 4. Section 5 discusses related work, and our conclusions and status are given in Section 6.

2. Version Data Model

What sets CAD data apart from conventional commercial data is an overriding need to model the complex hierarchical structure of designs (i.e., configurations) across time (i.e., versions). In our model, objects are internally denoted by the triple *name[i].type*, where *name* is the object name, *i* is the version number, and *type* is its design representation type, e.g., Adder[1].layout. Note that representation type and object-oriented type are orthogonal concepts (see Figure 2.1). Adder[1].layout is a layout object, but also it might be a functional unit object which is a kind of datapath object, etc. Besides configurations and versions, it is necessary to identify equivalent (or corresponding) portions of the design across different hierarchically structured representations.

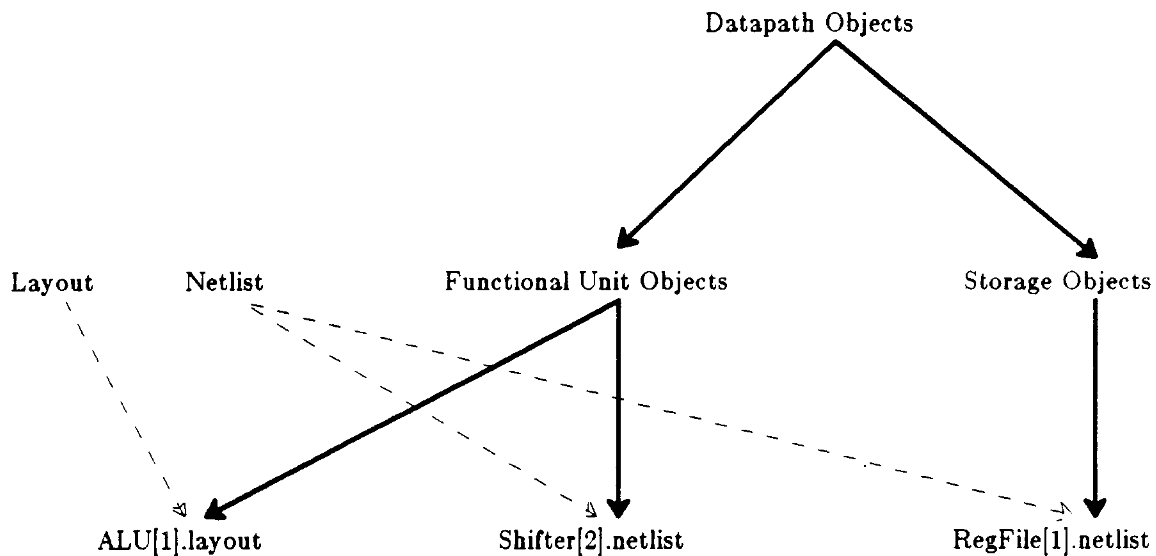


Figure 2.1 -- Type Hierarchy with Representation and User-defined Types

In CAD data, it is useful to make representation types, such as layout, netlist, etc., explicit. There is no real difference between these representation types and other user-defined types, such as the taxonomy of datapath objects being partitioned into functional unit objects and storage objects.

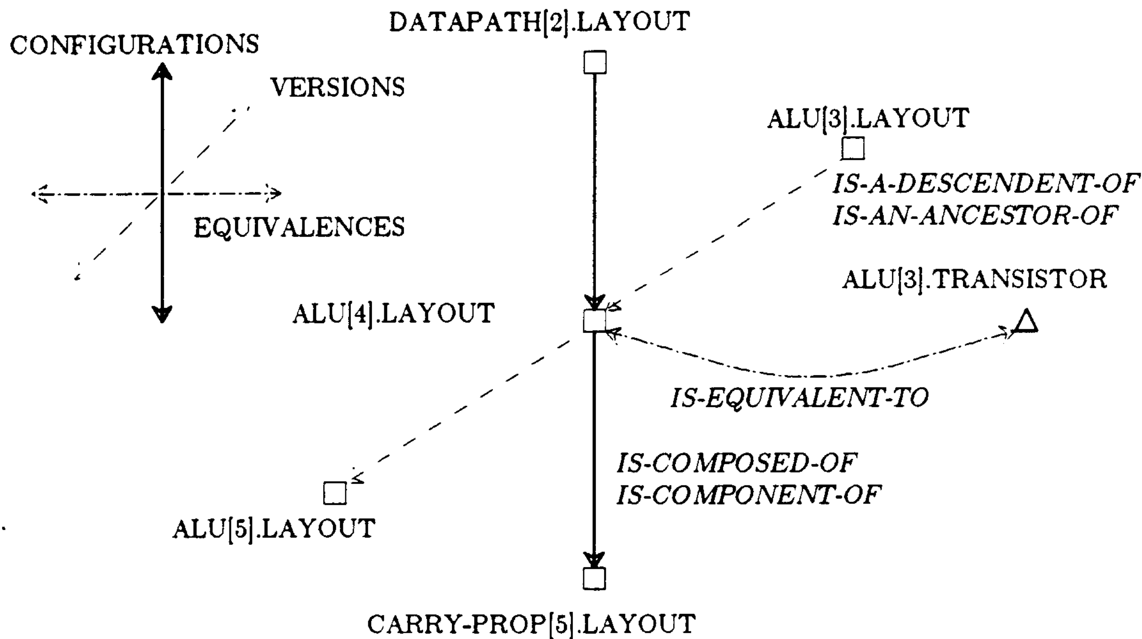


Figure 2.2 -- Version Data Model

Design data is organized as a collection of typed and versioned design objects, interrelated by configuration, version, and equivalence relationships. Only representational objects are shown. For example, ALU[4].layout is descended from ALU[3].layout and is the ancestor of ALU[5].layout. It is also a component of DATAPATH[2].layout and is composed of CARRY-PROPAGATE[5].layout. Additionally, ALU[4].layout is equivalent to other objects, such as ALU[3].transistor.

A key requirement for CAD applications is that the database system be able to support the definition, storage, and retrieval of complex collections of objects that can be treated as a single unit. These have many names, including *composite objects* [KATZ86a], *complex objects* [LORI83], *aggregation hierarchies* [ATWO85], and *molecular objects* [BATO84, BATO85a]. To support this, our model explicitly supports composite/component relationships among objects.

Objects also evolve over time, and another way of organizing objects is along *version derivation histories*. Coupling the concept of versions with composite objects leads to *configurations*. A configuration is a composite unit whose components are bound to specific versions. For example, a composite datapath object may consist of an ALU, register file, and shifter object components. A version of the datapath object becomes a configuration consisting of specific versions of the ALU, register file, and shifter objects.

The third and final kind of relationship among objects denotes *equivalences* across representations. For example, ALU[1].layout is equivalent to ALU[2].logic means that both objects are different representations of the same real world entity, the ALU. The model and its three distinguished structural relationships (configurations, version histories, and equivalences) are summarized in Figure 2.2. The Version Data Model uses these

distinguished *structural relationships* to logically organize objects within the design space and physically place objects on disk (see Section 4).

An operational model is needed in addition to the logical model. Objects are *checked-out* from shared archives into designers' private workspaces. The modified object can then be either returned as a new version to the shared archive or *checked-in* to a shared group workspace where the integration is made with other designers' work [KATZ87, KATZ86a,b]. The description of how a new instance is to inherit its definitions occurs at check-out time, when the instance is first created as a new version. We will have more to say about this in the next section.

3. Inheritance: What It Means

The purpose of inheritance is to propagate data definitions among database objects, making it possible to make a definition once and have it apply to many instances. The object-oriented type system usually constrains what can be inherited as well as which interobject connections can be used for inheritance paths.

As a basis with which to explain these concepts, we will use Smalltalk-80 as our canonical model of an object-oriented type system. Objects are uniquely identified instances belonging to types. Type definitions are represented by a combination of attributes (called class/instance variables) and operations (called methods). Attributes describe static properties, while operations describe dynamic behavior. It is also possible to associate structural relationships (as we do) and constraints (as [BUCH86] does) with types to be inherited by their instances, if the model supports them. Types are further arranged in a lattice, inheriting attribute and operation definitions in turn from their supertypes. A type can override or refine a definition inherited from its supertype.

Although it is not supported in Smalltalk-80, most object type systems allow instances to be members of more than one type. For example, in Figure 2.1, we have ALU[1].layout as an instance of both the *layout* and *functional unit* types. This introduces the well known problem of *multiple inheritance*, i.e., if an instance inherits the same information from more than one place, how can this be performed unambiguously. Although there are many proposals as to how to handle multiple inheritance, they all reduce to having the object definer disambiguate the inheritance for the system.

The model of inheritance as implemented in Smalltalk-80 has several desirable features. New object types can be introduced into the lattice without the type definer needing to know all aspects of its supertypes, since inherited definitions can always be overriden within the new type. Thus, information hiding and independence are supported. Further, new types (or instances) can be created with many of their properties and behavior inherited from their supertype (or type). Thus, inheritance provides defaults for new type definitions and creation of instances.

Some aspects of CAD data complicate the Smalltalk-80 style of inheritance, in particular, the introduction of versions and composite objects. The issue is how to embed these structures into type-instance models while extending the inheritance mechanisms to cover them. For example, all versions of an object could be modeled as instances of the same generic type [BATO85b]. Thus, certain properties and behaviors, common to all versions, can be defined in the type definition and inherited by each version instance.

However, there are some properties and behaviors (as well as structural relationships and constraints) that an offspring version might wish to inherit from its parent version

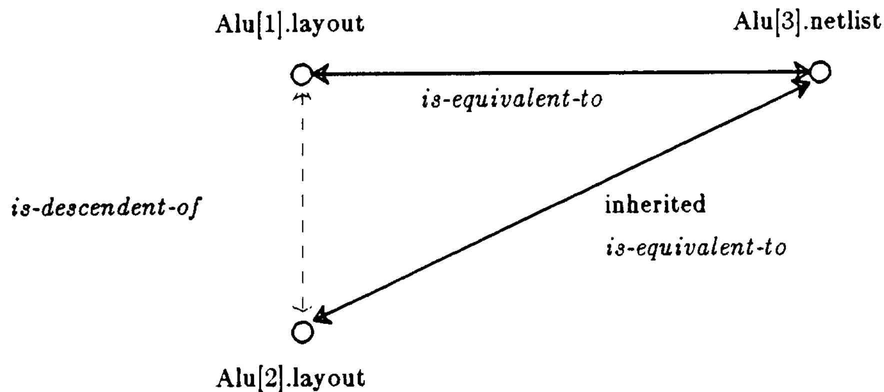


Figure 3.1 -- Inheriting Instance Specific Correspondences

Correspondence relationships constraint two objects to be equivalent. A new version typically inherits these relationships from its immediate ancestor. This is a good example of inheritance along version relationships which are not easily modeled with conventional type-instance inheritance.

directly rather than from its type. Consider the following example. If ALU[2].layout is equivalent to ALU[3].netlist, then a new descendant of ALU[2].layout should inherit this relationship as a default (see Figure 3.1). While it is possible to constrain all ALU layout versions to be equivalent to some ALU netlist instance, it is not possible to specify specific correspondences on an instance-by-instance basis without introducing a proliferation of types (see Figure 3.2).

We can envision cases where it is desirable to inherit along relationships other than version histories. For example, [BUCH86] describes how constraints on a composite object can be inherited by its components. In fact, it should be possible to inherit information along any kind of relationship known to the system, be it type-instance, ancestor-descendent, composite-component, or even among equivalents. Standard type-instance inheritance cannot model these kinds of information propagations by itself. While it is possible to implement it using user-defined operations that can compute what to inherit at run-time, it is not desirable to do so, since the inheritance semantics is not obvious from the data model and cannot be exploited by the database system. Therefore, we propose to support direct *instance-to-instance* inheritance.

Of course, instance-to-instance inheritance leads to even greater complexity in terms of disambiguating multiple inheritance. Consider the version model introduced in the last section. If a new instance is to inherit from a containing composite object or an equivalent object, it is necessary to establish the relationships with these as part of a check-out context. The format of the context-dependent check-out command is as follows:

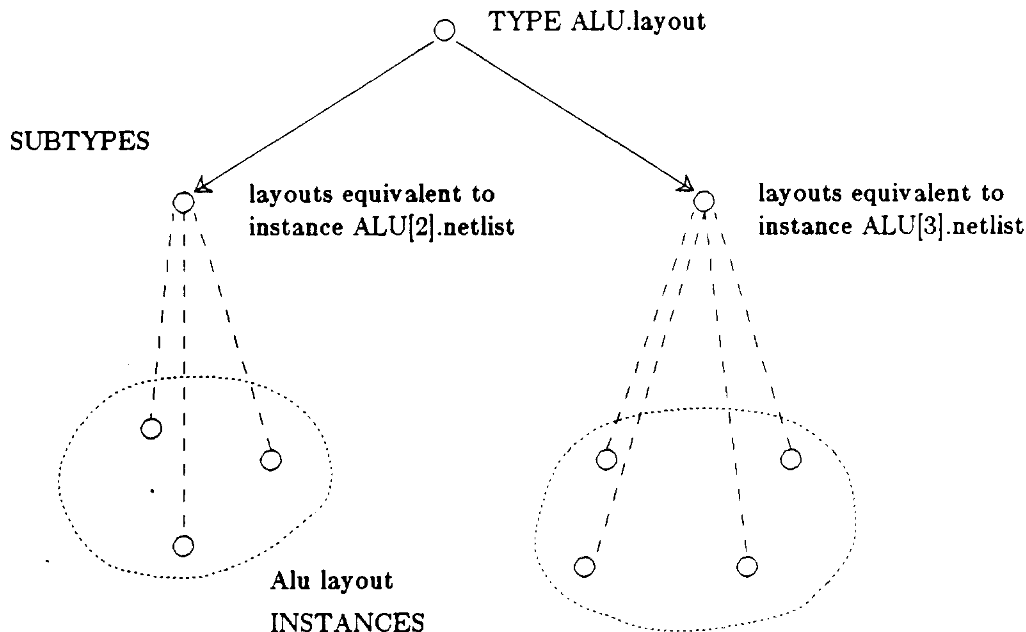


Figure 3.2 -- Handling Instance-to-Instance Inheritance with Subtypes

A subtype is created for each group of instances that shares a common attribute or constraint. Obviously this leads to an undesirable proliferation of subtypes.

check-out <object-name>
  equivalent-to {<list of equivalent instances>}
  contained-within {<list of composite instances>}
  {list of the following:
    <attribute-name> like
      [GENERIC, ANCESTOR, EQUIVALENT, COMPOSITE]};

(NOTE: "GENERIC" is our terminology for the type of all versions of an object, e.g., the type ALU.layout is the GENERIC of all ALU[i].layout instances). For example, assuming that ALU layouts have three attributes, *input-port, output-port,* and *behavior,* the following check-out command:

check-out ALU[3].layout
  equivalent-to ALU[2].netlist,
  contained-within Datapath[2].layout,
    input-port like ANCESTOR,
    output-port like COMPOSITE,
    behavior like EQUIVALENT;

would create the version ALU[4].layout of Figure 1, whose input-port attribute would be inherited from ALU[3].layout, whose output-port would come from Datapath[2].layout, and whose behavior attribute would be from ALU[2].netlist. While it looks laborious to

specify what should be inherited, note that the default is to have exactly the same inheritance behavior as the instance's immediate ancestor. The detailed specifications are only necessary if the default inheritance is to be overridden. The order of the instances within the equivalent-to and contained-within lists is used to disambiguate multiple inheritances: the first object found within the list that has the described attribute will furnish its value in the new instance. The LIKE clause can mention a specific instance name if this simple form of disambiguation is not sufficient.

The key implementation challenge for all inheritance mechanisms is how to propagate a changed attribute to all instances that inherit it. Inheritance can be implemented *by copy* or *by reference*, i.e., the inherited value can be cached with the instance or the inheriting instance can refer to the type (or source instance) that defines the value. The former approach has the advantage of fast access, but slow propagation of changes (i.e., the change must be propagated to all cached instances). The latter has better update performance since changes are made in only one place, but exhibits slow access. Note that in either case, it will be necessary to navigate across inheritance links, be it from instance to attribute definer in an attribute access or from the attribute definer to instance in a change of definition.

Why have systems preferred to implement type-instance rather than instance-to-instance inheritance? The answer is due primarily to ease of implementation. In type-instance inheritance, if inheritance is implemented solely by reference, then these references are restricted to the type hierarchy, whose definitions can reside in a system catalog. Thus, type-instance inheritance can exploit the known location of type information to efficiently resolve the inheritance. Instance-to-instance inheritance presents more complex implementation challenges, which will be described in the next section.

4. Instance-to-Instance Inheritance: Efficient Implementation

4.1. Introduction

In all actual systems known to us, conventional inheritance of attributes is performed at instance creation time while operation look up is done at run-time. However, instance-to-instance inheritance allows attributes to be selectively inherited at run-time. To achieve good performance, this run-time flexibility requires more sophisticated approaches for clustering and buffering. By choosing an appropriate clustering strategy, the system can minimize the number of I/Os required to traverse inheritance links, either to satisfy attribute accesses or to propagate changes. In this section, we discuss issues related to complex object clustering and smart buffering. We also address how to perform on-line backup of the richly interconnected design space.

4.2. Smart Clustering

Clustering is a well-known technique for improving referential locality. The objective is to place frequently co-referenced objects near each other in physical memory. The partitioning of objects into clusters on disk can be done either statically, with the system quiesced, or dynamically, as the system is running. We focus on the latter approach in this section.

4.2.1. Algorithm Overview

Many computer-aided design applications frequently use configuration relationships, e.g. a design rule checker must walk the configuration from leaves to root as it performs its checks. However, most inheritance references are along version history relationships, since a descendant version typically obtains information from its ancestor. This observation illustrates the difficulty in capturing access patterns for use by the clustering algorithm. In our algorithm, the inputs include users's hints (e.g., access by configuration vs. access by version), the structural relationships among objects, interobject access frequencies, and the characteristics of inherited attributes.

The system could decide how to cluster instances based on instance specific statistics. Unfortunately, statistics collection on an instance specific basis is simply too expensive. Rather, interobject access frequencies are described at the type level for each kind of structural relationship, e.g., 20% of accesses from ALU layouts are along version relationships, 75% along configuration relationships, and 5% along equivalences. The user specifies these frequencies at data type creation time. Should the frequencies change at a later time, existing instances will not be reclustered, but the clustering strategy will be affected for new instances.

Every instance inherits its access pattern frequencies at the time of its creation. If the object could be an instance of more than one type, the user must disambiguate from which type the frequencies are to be inherited from. Note that these inherited attributes are always implemented by reference, i.e., no space is allocated at the instance level for them. For the access frequencies described in the previous paragraph, the initial placement of a new instance is likely to be on the same page with its composite object because the frequency of accessing configuration relationships is higher than others. Similarly, the target object will be placed on the same page with its ancestor object if the frequency of accessing version history relationship is the highest.

The algorithm can be sketched as follows (the full algorithm is presented in Section 4.2.2). For each newly created instance, the clustering algorithm chooses an initial placement based on which of the instance's relationships is most frequently traversed. Since there is a tight interplay between access frequencies and choice of how to implement inherited attributes, the algorithm also chooses between implementation by copy and by reference for inherited attributes using an additional set of cost formulas. The augmented access frequencies (i.e., relationship traversal frequencies plus inheritance traversal frequencies) may change the initial placement choice.

A further complexity is introduced if there is insufficient space for the instance in its preferred location. The clustering strategy actually chooses an ordered collection of candidate pages on which to place the new instance. If the preferred candidate is full, the storage manager must either split this page to make room for the instance, or it must choose the next best candidate page which has space. The page is split if the expected access cost resulting from the split is an improvement over putting the new object in the next best candidate page. Otherwise, the next candidate page is examined, and the decision process recurses if there is insufficient room on it.

Estimating the cost resulting from the page-split can be formulated as a graph partitioning problem. Suppose object A inherits attribute X from object B (or there is a structural relationship between A and B). There will be an arc from A to B where the arc value is the cost of the run-time lookup for attribute X and the node capacity will be the

object size (similarly for the cost of traversing the relationship). However, graph partitioning is known to be NP-complete and is not suitable for run-time clustering. Instead, we use a greedy algorithm that partitions the nodes of the inheritance-dependency graph into two subsets that can fit into a page individually. At the same time, the greedy algorithm tries to minimize the total cost of broken arcs. Because it does not try to find the optimal partition and only scans through the set of arcs once, the total running time is guaranteed to be linear.

4.2.2. Algorithm Details

The clustering algorithm is presented in pseudo code in Figure 4.1. The procedure *cluster_object()* is called when the system wants to do clustering for a target object. The target object could be a newly created object or an updated object. That is, if the change is a structural or inheritance update to the target object, the system also calls *cluster_object()* to do reclustering based on the latest statistics. Note that while clustering decisions are based on type-level statistics, choice of inheritance implementation is based on instance-level update statistics (see Step 1 below).

For every step in the *cluster_object()*, a detailed description follows:

step 1: Before choosing a candidate page to place the target object, the algorithm needs to know the clustering policy as defined by users. The choices are either to split the candidate page if it is out of space or to choose the next best page instead.

By examining all the inherited attributes of the target object, the algorithm determines the implementation strategy for every inherited attribute. For each attribute, a one-byte counter is used to monitor the update frequency. When the counter is overflowed by the number of updates, it is permanently set to be 255 (i.e., 2^8). Using this one-byte counter, the system can make inheritance implementation decision on an instance basis. If an inherited attribute has been updated frequently, *by reference* implementation is used. The "update threshold" can be set by the type definer. Otherwise, the algorithm will use *by copy* to implement this inherited attribute. These implementation decisions are made by calling *get_by_copy_set()* and *get_by_ref_set()*, and the results are returned in *copy_set* and *ref_set*.

Furthermore, the system needs to determine the placement strategy within the physical space. All pages which contain source instances for these inherited attributes are returned in *inh_page_set* by calling *get_all_inh_page()*. Similarly, pages which contain the target object's interrelated objects such as its ancestor instance and component instances, are returned in *struct_page_set* by calling *get_all_struct_page()*. The union of *inh_page_set* and *struct_page_set* creates a set of candidate pages for placement in later steps.

step 2: If a by-reference attribute is not placed in a chosen candidate page **p**, the system needs to dereference it at run-time. So an effort is made to place the instance on the same pages as the sources of its inherited attributes, taking account of their access frequencies. Therefore, the cost of storing a target object in page **p**, modeled by *Ref_LookUp(* **p** *)*, is incremented by weight(p), which is a function of the access frequency of structural relationship for page **p**.

step 3: For inherited attributes chosen to be implemented *by copy*, the system can either copy it to a candidate page **p** or needs to look it up at run-time. Therefore, we

```
PROCEDURE cluster_object(target_object)
BEGIN
        /* step 1: get initial information */
        cluster_policy := get_policy();          /* Is page splitting enabled? */
        copy_set := get_by_copy_set();           /* Inherited attributes implemented by copy.*/
        ref_set  := get_by_ref_set();            /* Inherited attributes implemented by reference.*/
        inh_page_set := get_all_inh_page();  /* Source pages for inherited attributes.*/
        struct_page_set:= get_all_struct_page();/* Source pages for structural objects.*/
        page_set := inh_page_set + struct_page_set;
        /* step 2: calculate ref_set lookup cost for each page */
        FOR p IN page_set                        /* If by-reference attribute r is         */
                FOR r IN ref_set                 /* not in page p, storing target object */
                        IF r NOT_IN p            /* in page p requires one run-time      */
                        BEGIN                    /* lookup for attribute r.               */
                             weight(p) := 1/(prob(p,struct_rel));
                             Ref_LookUp(p):= Ref_LookUp(p)+weight(p);
                        END;
        /* step 3: calculate copy_set lookup and storage cost for each page */
        FOR c IN copy_set                        /* If by-copy attribute c is not in page*/
                FOR p IN page_set                /* p, we could either cache it in page p*/
                        IF c NOT_IN p                    /* or change it implementation to be */
                        BEGIN                    /* by-reference.                         */
                             weight(p) := 1/(prob(p,struct_rel));
                             Copy_storage(p) :=Copy_storage(p)+sizeof(c);
                             Copy_LookUp(p):= Copy_LookUp(p)+weight(p);
                        END;
        /* step 4: calculate total cost of every page. If by-copy attributes are       */
        /*         implemented by reference, the total cost of storing target object   */
        /*          in page p is represented by Total_cost(p,1).  Otherwise, the cost  */
        /*          is represented by Total_cost(p,2).                                                  */
        FOR p IN page_set
                Total_cost(p,1) := Ref_LookUp(p)*Lookup_cost + Copy_LookUp(p)*Lookup_cost;
                Total_cost(p,2) := Ref_LookUp(p)*Lookup_cost + Copy_storage(p)*Storage_cost;
        /* step 5: pick up best candidate page and try to insert the object */
        candidate_page := Minimum (Total_cost);
        IF (cluster_policy EQ no_split)
                WHILE (NOT_FIT(candidate_page)
                        candidate_page := Next_Min (Total_cost);
        IF ( (cluster_policy EQ page_split) AND ( NOT_FIT(candidate_page))
                Split_page(candidate_page);
END;
```

Figure 4.1 -- Psedo Code for cluster_object

have two cost variables: *Copy_storage(* **p** *)* is used to model the cost of storage and is incremented by the size of the by-copy inherited attribute, whereas *Copy_LookUp(* **p** *)* is used to model the cost of by-reference implementation and is incremented by weight(p) as with *Ref_lookup(p)*. The latter cost component is of interest because it models the cost of propagating a change to the instance, which is reduced if the instance and the source of the inheritance are placed on the same page.

step 4: To determine the candidate page, the system needs to transform the costs (i.e. lookup and storage costs) into the same scale for comparison. The lookup cost *Lookup_cost* is represented by $P_{hit}*C_{buf}$ + $(1\text{-}P_{hit})[C_{os}+P_{io}*C_{io}]$ where P_{hit} is the probability of buffer hit, P_{io} is the probability of doing I/O during buffer replacement, C_{io} is the cost of I/O, C_{buf} is the cost of searching through buffers, and C_{os} is the cost of getting free page from O.S. The storage cost *Storage_cost* is represented by *Lookup_cost* * scale_factor* where *scale_factor* is determined by users.

step 5: If the clustering policy does not permit page splits and the candidate page is out of space, the clustering algorithm chooses the next minimum cost candidate page to insert the target object. Otherwise, the system tries to split the candidate page to minimize the new run-time lookup cost.

The *Split_page()* procedure in step 5 tries to minimize the look up cost from the page-split. If a page contains N inherited attributes, its look up cost is represented as N * *Lookup_cost* which is described in detail in step 4. The detailed algorithm for procedure *Split_page()* is shown next:

Page_split Algorithm: Assume that the arc costs C_{e_i} (i.e. run-time lookup cost) are always maintained and sorted in the page header. The node capacity Cap_{v_j} (i.e. the object size) is available from the **object header** which is maintained by the system. Subset A and B represent the sets of objects assigned to the new pages after splitting. Both subset A and B are empty at the beginning and the available capacity of A and B are set to be (maximum_page_size * 0.75).

(1) Select the maximum value arc from E as e_{target} and set E to be (E - $\{e_{target}\}$). Let v_{head} and v_{tail} to be the head and tail nodes of arc e_{target}.

(2) Supposed both v_{head} and v_{tail} are new to subsets A and B. Insert v_{head} and v_{tail} into subset A if $Cap_{v_{head}}$ plus $Cap_{v_{tail}}$ is less than the remaining capacity of subset A. Otherwise, insert v_{head} and v_{tail} into subset B if subset B has space for these nodes. If neither subset A or B could accommodate both v_{head} and v_{tail}, a broken arc is found and $C_{e_{target}}$ is added into C_{total}.

(3) Supposed v_{head} is in subset A and v_{tail} is not in subset A or B. Insert v_{tail} into subset A if feasible. Otherwise, a broken arc is found and $C_{e_{target}}$ is added into C_{total}.

(4) Supposed both v_{head} and v_{tail} are visited before, a broken arc is found and $C_{e_{target}}$ is added into C_{total}.

(5) Loop back to (1) until arc set E is empty.

Algorithm Analysis: This algorithm is greedy. Since we only scan through the edges once, the total running time is guaranteed to be O(n) where n is the number of edges in G.

A page split example is shown in Figure 4.2. The clustering algorithm described in Figure 4.1 has chosen a candidate page shown in Figure 4.2 to store the target object X. Supposed that the maximum page size is 4000 bytes and the clustering policy is to split the page when it is out of space. Because the target object X is too large to fit into the candidate page, the clustering algorithm invokes the page splitting algorithm to determine a new page layout for these objects. The page splitting algorithm first places object B and D in the same page since the look up cost from B to D is the largest one. Object E is then chosen to be on the same page since arc cost from B to E is the next largest. However, object A and F are placed in another page due to the limited page capacity. After scanning through the arcs in the inheritance dependency graph, the page splitting algorithm produced the final page layout shown in Figure 4.3. Notice that the look up cost of the

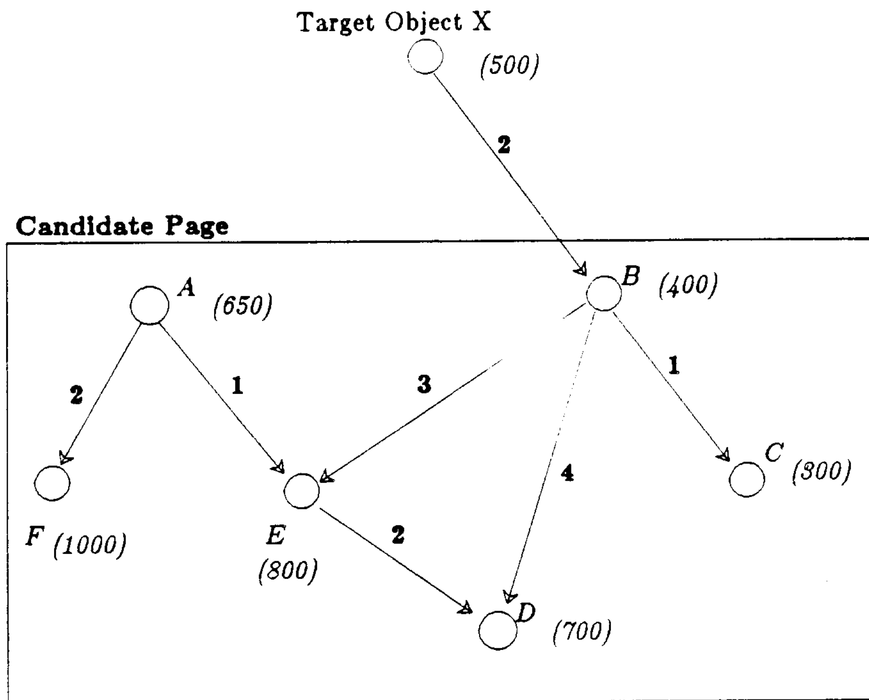


Figure 4.2 -- Page Split Example

Each node represents an object and the number associated with it is the size of the corresponding object. The arc represents either the inheritance dependency or structural relationship, and the number associate with it is the run-time look up cost. All the nodes within the box is on the same page and the target object X is too large to fit into the candidate page in this example.

new page layout is 1 (i.e. from object A to E) whereas the look up cost without splitting is 2 (i.e. from object X to B). Therefore, splitting the candidate page has reduced the look up cost by 1.

4.2.3. Discussion

To avoid the extra cpu time caused by the clustering mechanism, users may disable automatic reclustering when the system is heavily loaded. Or they may enable/disable clustering based on the characteristics of their operations and data. For instance, operations like check-in/out require a large amount of data to be inserted into workspaces. Enabling clustering will provide better response time in the future at the expense of extra

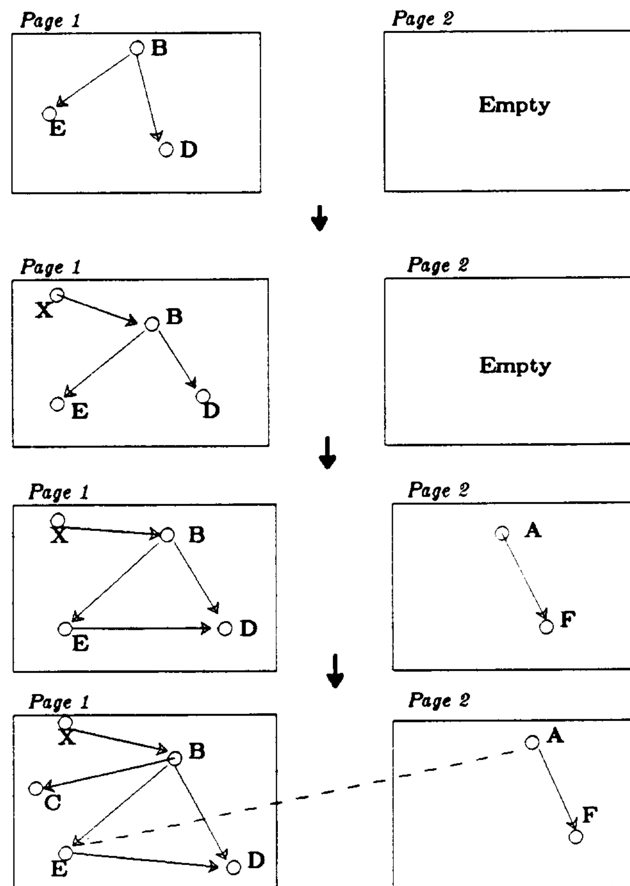


Figure 4.3 -- New page layout after page splitting

After page splitting, objects A and F are in page 2 and the rest of objects are stored in page 1. The dotted line between object A and E represents the new run-time look up cost after page splitting.

cpu time during check-in/out. Users can therefore tune the context-dependent clustering policy to provide reasonable response time for their applications.

4.3. Smart Buffer Replacement

The Sun Benchmark proposed by [CATT87] clearly states that response time is more important than throughput in computer-aided design applications. The most common operations of CAD tools are navigation along the structural relationships and simple retrieval of design objects. Unfortunately, these operations are poorly supported or inefficiently implemented by most commercial database systems. Therefore, an object-oriented buffer manager differs from its counterpart in a conventional database system in the following ways: (1) the focus is on response time rather than throughput, and (2) extra semantics can be exploited by the system.

To obtain better response time, the buffer manager must exploit knowledge about the structural and inheritance relationships among objects. This can be used to determine prefetching and buffer replacement strategies. Response time is improved if appropriate objects can be prefetched before actually being accessed, or if related objects can be kept in the buffer pool even if the relationships span disk pages. For instance, if buffer X contains an object that references some attributes in buffer Y through an inheritance link, the buffer manager should try to keep buffers Y and X in the buffer pool at the same time. This kind of information cannot be exploited unless it is made known to the system, and this is the primary rationale for why we have distinguished the three kinds of relationships among objects.

4.3.1. Algorithm Overview

The implementation overview of an object-oriented buffer manager is presented here. An unsophisticated buffer manager uses a simple LRU buffer replacement policy and attempts no prefetching. A more sophisticated approach uses a priority scheme such that the lowest priority pages are the ones to be replaced first. The key challenge is to use the semantics of the interrelationships among objects on the buffered pages and hints about the access patterns to set the priorities intelligently. Frequently accessed pages have their priority increased. Infrequently accessed pages have their priority reduced, but this may be modified by their interrelationships with other pages, especially if those are frequently accessed. Whenever an object is accessed, its related pages (e.g., pages containing an object's components and its inherited attributes) might be in buffer pool already. Traditional LRU buffer replacement, which has no knowledge about these interrelationships, could easily choose these pages to be replaced and thus invokes extra I/Os to bring them in later.

Besides intelligent buffer replacement, another way to obtain good response time is to be smart about prefetching. At the beginning of an interaction with the database, the users provide the buffer manager with access hints, such as "my primary access is via configuration relationships". This information influences the buffer manager's prefetch strategy. Touching an object causes the page containing it and the pages containing its immediate subcomponents to be brought into the buffer pool and given the same high priority. This achieves extremely good performance for applications that walk the configuration hierarchy. Similar prefetch hints can be used to obtain a version object, its immediate ancestor, and its immediate descendents. Also, equivalence relationships can be used to obtain all objects equivalent to the one being accessed. Inheritance is treated in a

similar fashion for determining prefetch groups.

4.3.2. Algorithm Details

For every *fetch object* operation, users can provide access path information and prefetch hints along with the target object identifier. The pseudo code of procedure *Fetch_Object()* is shown in Figure 4.4 and a detailed description follows:

step 1: If the target object is in the buffer pool, the buffer manager sets the priority of the containing buffer to be **HighPri**. Otherwise, the buffer manager allocates a buffer frame, fetches the object into it, and sets the priority to be **HighPri**.

step 2: To avoid useful pages being replaced, the buffer manager tries to locate all buffer frames which contain either structural or inheritance information which are related to the target object, and sets these buffer frames to be **HighPri**.

step 3: If the prefetch option is enabled, the buffer manager does prefetching for all the structural and inheritance related objects on behalf of the target object.

To illustrate more on these steps, an example is shown in Figure 4.5. Supposed the target object X is located at page A and its component objects are in page B, C, and D. The prefetch option of this Fetch_Object() call is disabled and the structural_rel parameter is specified as *configuration hierarchy*. Before calling the Fetch_Object() to fetch the target object X, the buffer pool already had pages B and D which are ready to be replaced because of their low priority status. To fetch object X into buffer pool, the buffer manager first allocates a free buffer. Since both page B and D have low priority status, either one of them could be chosen to be replaced. However, this decision is not desirable because object X may reference these pages very soon. To avoid this, the *Fetch_Object()* procedure considers information provided by callers, e.g., the *structural_rel* parameter is set to be *configuration hierarchy*, and contents of all buffers, e.g., page B and D contain component objects of object X. It trie to keep page B and D in the buffer pool by increasing their priority status. The final buffer pool layout after this *Fetch_Object()* call is also shown in Figure 4.5.

4.3.3. Discussion

Note the close interplay between the clustering algorithm and buffer management. If the clustering algorithm has done a good job, then interrelated objects will be placed on the same page or in a small collection of pages. If not, and if access along these relationships are frequent, then the clustering algorithm will adapt to the access patterns by reorganizing the placement of objects. The buffer manager can alert the clustering algorithm about the need to reorganize.

4.4. Other Opportunities: On-line Backup

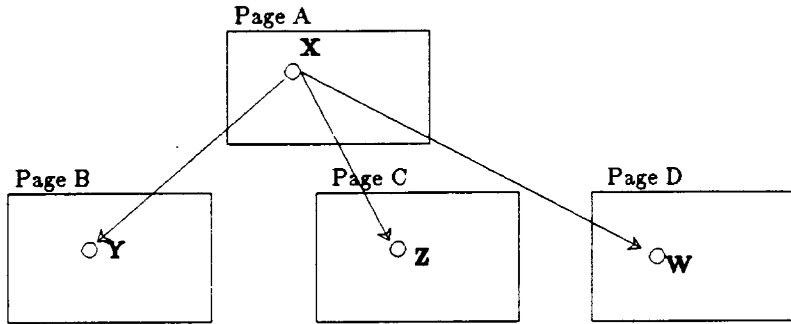
The structural relationships of Section 2 form a very complex web of interconnections among design objects. Added to this complexity is our proposed inheritance semantics, yielding a very complex design space indeed. This presents a challenge to all existing database utilities, especially the on-line backup service. We will concentrate on its implementation difficulties in this section.

As the database ages, older versions become less frequently accessed as ever more recent versions are added. Some method for migration of very old versions to archival storage is necessary. Back-up is usually necessary when the system runs short of disk

```
PROCEDURE Fetch_Object(ObjID, structural_rel, prefetch)
BEGIN
        /* step 1: fetch the base page of target object. */
        IF (ObjID NOT_IN BufferPool)
        BEGIN
                buffer_id := Get_bufferframe();
                Get_page(ObjID, buffer_id, HighPri);
        END;
        /* step 2: get structural and inheritance links used by the target object.  */
        /*      If their containing pages are in buffer pool, set their priority    */
        /*      to be HighPri.                                                       */
        structural_links := Get_struct_links(ObjID, structural_rel);
        FOR link IN structural_links DO
                IF (link IN BufferPool)
                        Set its bufferframe to be HighPri;
        inh_links := Get_inh_links(ObjID);
        FOR link IN inh_links DO
                IF (link IN BufferPool)
                        Set its bufferframe to be HighPri;
        /* step 3: If prefetch option is on, prefetch all related pages into memory. */
        IF (prefetch)
        BEGIN
                FOR link IN structural_links DO
                        IF (link NOT_IN BufferPool)
                        BEGIN
                                buffer_id := Get_bufferframe();
                                Get_page(link, buffer_id, HighPri);
                        END;
                FOR link IN inh_links DO
                        IF (link NOT_IN BufferPool)
                        BEGIN
                                buffer_id := Get_bufferframe();
                                Get_page(link, buffer_id, HighPri);
                        END;
        END;
```

Figure 4.4 -- Pseudo Code for Fetch_Object

Target Object Configuration



Buffer Pool before fetch_object(X)

	Page B lowpri	Page G		- - - - -	Page D lowpri		Page U		

New layout after fetch_object(X) without prefetching

	Page B High pri.	Page G	Page A High pri.		Page D High pri.			Page T

Figure 4.5 -- Smart Buffer Management Example

Each box represents a page frame and the circle within a box represents an individual object. The arc between circles is the configuration relationship. In this example, object X is composed of objects Y, Z and W. The replacement priority is represented as "low pri" and "high pri". Two buffer layouts are shown to illustrate the difference before and after calling Fetch_Object().

space, or to provide a reliable copy of the design space to protect against media failure, or when a new design release obsoletes an older one. When backing up a design object, the system needs to ensure that the structural relationships that reference the object are well maintained. For instance, consider the case where ALU[3].layout is composed of Adder[1].layout and Reg[2].layout, and Adder[1].layout was backed up and is no longer available on disk. When Adder[1].layout is referenced by ALU[3].layout later on, the system should be able to return meaningful information (e.g. "Referred Object is not online"). At the same time, designers should continue to be able to browse through the structural relationships of the design space, i.e., although some design objects are backed up, the corresponding structural relationships should still be visible. Later on, when backed-up objects are restored to the design space, the system should repair the references

appropriately.

For instance-to-instance inheritance, the backup situation is more complicated, since other objects depend on the backed-up object to obtain their inherited attributes. It is not sufficient to return an error message in this case. The back-up manager must choose between caching the referenced attribute in the instances or it could simply avoid backing up the object in question. The usual clustering and access formulas come into play in making this decision.

Algorithms like the backup manager need frequent and fast access to schema information, to be able to interpret the attribute types and to traverse connections among objects. To help it deal effectively with the interconnected design space, a special internal data structure is used. The basic idea is to distribute the schema information (e.g., name of the attributes, offset, and type) among the objects to make objects self-describing. Each attribute of an object has an attribute descriptor that contains backup status (if inherited), attribute type, and offset information. This is advantageous for fostering the locality of schema lookup. Using encoding techniques, the information describing structural relationships and inheritance links need not take up much space on the page.

5. Related Work

Existing object-oriented database management systems, either prototype or product, have implemented only very primitive object-based clustering mechanisms [MAIE86, ATWO85, ZDON84, KIM 87]. The common characteristics of these implementations are: (1) a segment is the unit of clustering, and (2) user's hints can be used at object creation time. For instance, users may provide hints such as "place near object XX". The system then tries to store the target object with object XX, in the same or an adjacent page. Since these systems do not model structural relationships as first class object in their data models, the storage component has no information to exploit during the clustering process. That is, users' hints are the only useful semantics which can be used by the storage component. Moreover, these systems do not perform reclustering when object structures are changed, e.g., when new components are included in a composite object. Neither do they consider the clustering effects due to inheritance.

6. Summary and Conclusion

The object-oriented approach is a good match for computer-aided applications. However, traditional inheritance semantics do not model all of the information propagation requirements of VLSI design environments. In this paper, we have shown examples where a new inheritance semantics, instance-to-instance, is better matched to the needs of computer-aided design applications.

Instance-to-instance inheritance introduces new implementation challenges in the areas of physical database design, buffer management, and design back-up. We have sketched new adaptive clustering algorithms, and have indicated how semantic information and user hints can be exploited for more intelligent buffer management. There is important an interplay between these activities: the buffer manager's performance will suffer from a bad clustering, but it can alert the clustering algorithm about changes in access patterns to rectify the situation. We also investigated how the proposed inheritance semantics affect applications such as on-line backup of the design space.

At the present time, the algorithms described in this paper are being evaluated and will be implemented in a next generation Version Server [KATZ86b] for computer-aided design data currently under design at U.C. Berkeley. We wish to acknowledge NSF grant MIP-8706002, which is supporting this new work, and NSF grants MIP-8352227 and MIP-8403004 which made it possible to build the initial Version Server prototype.

7. Reference

[ATWO85] Atwood, T., "An Object Oriented DBMS for Design Support Applications," Proc. IEEE COMPINT 85, Montreal, Canada, (Sept. 1985).

[BATO84] Batory, D., A. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models," 10th Intl. Conf. on VLDB, Singapore, (Aug. 1984).

[BATO85a] Batory, D., W. Kim, "Modeling Concepts for VLSI CAD Objects," ACM Trans. on Database Systems, V 10, N 3, (Sept. 1985).

[BATO85b] Batory, D., W. Kim, "Supporting Versions of VLSI CAD Objects," M.C.C. Technical Report, Austin, TX, (1985).

[BORN86] Borning, A., "Classes vs. Prototypes in Object-Oriented Languages," Proc. FJCC, Dallas, TX, (Nov. 1986).

[BRAC83] Brachman, R. J., "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks," IEEE Computer Magazine, (October 1983).

[BUCH86] Buchmann, A., R. Carrera, M. Vazquez-Galindo, "A Generalized Constraint and Exception Handler for an Object-Oriented CAD DBMS," Proc. 1986 Intl. Workshop on Object-Oriented Database Systems, Asilomar, CA, (September 1986).

[CATT87] Cattell, R.G.G. et al. "Benchmarking Simple Database Operations," Proc. ACM SIGMOD Conference, San Francisco, CA, (May 1987).

[KATZ87] Katz, R. H., R. Bhategja, E. Chang, D. Gedye, V. Trijanto, "Design Version Management," *I.E.E.E. Design and Test Magazine*, V 4, N 1, (February 1987).

[KATZ86a] Katz, R. H., E. Chang, R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Database," ACM SIGMOD Conf., Washington, DC, (May 1986).

[KATZ86b] Katz, R. H., E. Chang, M. Anwarrudin, "A Version Server for Computer-Aided Design Databases," ACM/IEEE 24th Design Automation Conf., Las Vegas, NV, (June 1986).

[KIM 87] Kim, Won, et. al., "Composite Object Support in ORION-1," Proc. OOPSLA'87 Conf., Orlando, FL, (Oct. 1987).

[LIEB86] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in

Object Oriented Systems," Proc. OOPSLA'86 Conf., Portland, OR, (Sept. 1986).

[LORI83] Lorie, R., W. Plouffe, "Complex Objects and their use in design transactions," Engineering Design Application Proceeding from SIGMOD Database Week, (May 1983).

[MAIE86] David Maier, Jacob Stein, Allen Otis, Alan Purdy, "Development of an Object-oriented DBMS," Technical Report CS/E-86-005, (April, 1986).

[MITT86] Mittal, S. J., D. G. Bobrow, K. M. Kahn, "Virtual Copies: At the Boundary of Between Classes and Instances," Proc. OOPSLA'86 Conf., Portland, OR, (Sept. 1986).

[ROWE86] Rowe, Larry, "Shared Object Hierarchy," Proc. of 1st International Workshop on Object-oriented DBMS, Pacific Grove, CA, (Sept. 1986).

[ROWE87] Rowe, L., Stonebraker, M., "The POSTGRES Data Model," 13th Very Large Database Conference, Brighton, England, (Sept. 1987).

[ZDON84] Zdonik, S. B., "Object Management System Concepts," Proc. 2nd SIGOA Conf. on Office Information Systems, Toronto, Canada, (June 1984).