A Conceptual Model for Dynamic Clustering in Object Databases

Qing Li*
Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong
qing@uxmail.ust.hk

John L. Smith
Division of Information Technology
CSIRO
Canberra, ACT 2601, Australia
smith@csis.dit.csiro.au

Abstract

In object-oriented database systems, it is assumed silently that fundamental object types and inter-object relationships can be classified statically, prescribing basic structural and behavioral properties for all the objects in the database. Such a classification-based approach falls short of supporting those data-intensive applications requiring more advanced dynamic functions. A particular kind of such advanced functions is "dynamic clustering" – the ability to group/cluster existing objects to form typeless, ad-hoc collections (called "clusters") which are directly denotable and employable. Such clusters can be formed by taking copies or by including the Oids of the input objects, and can be explicitly defined or indefinitely defined, exhibiting thus different impacts and characteristics, ranging from loosely-coupled to tightly-coupled ones. In this paper we describe an on-going project of devising a conceptual clustering model. A real life DB application is used as a basis for testing and evaluating the model. An implementation prototype has been implemented, based on a comprehensive object database programming system.

*The work by this author was supported, in part, by the Commonwealth Scientific and Industrial Research Organization (CSIRO) under contract BT00825.

**Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992**

1 Introduction

Object-oriented languages and systems are becoming more and more popular for applications which model environments that lend themselves to extensive classifications, complex objects and inter-object relationships [KLE89, Mey88, ZME90]. In a conventional object-oriented database, the conceptual structure (schema) is embodied by a collection of abstract data types (called "classes" in this paper) which, when defined, are organized into an inheritance (ISA) hierarchy. Objects can then be created within classes, instantiating and justifying these predefined data types. In this way, a class prescribes both structural and behavioral properties (attributes and methods) of its objects, and objects in the class can be stored efficiently through a shared representation. Further, set-oriented access of class members can be applied efficiently, which is suitable and desirable for many applications whose objects can be classified statically (or at least to a large extent, with additional facilities for introducing/deriving new classes such as specialization and/or generalization [Sci89, SN88]).

Such a classification-based approach, however, falls short of supporting those applications involving objects and inter-object relationships that are by nature tentative, irregular, ill-structured, evolving, or simply unpredicatable. In particular, certain objects may arise dynamically as "clusters" of existing objects in some *ad hoc* fashion. While it is always in theory possible to accommodate such new types of objects through introducing new classes accordingly, in practice there are several reasons against this obvious approach. First, the number of such *ad hoc* objects may not be large enough to warrant the introduction of new classes (e.g., we may end up with many *ad hoc* classes). Second, the evolving/uncertain nature of some objects implies the inappropriateness of using class structure (as the classes would have to undertake similar evolution structurally). Third, some of the objects may be introduced merely for tentative/temporary purposes (e.g., for comparison of different alternatives), thus it would be too costly to introduce classes for such temporary objects. Finally, we feel that there is some deficiency in the current object class paradigm when modeling intra-class object (behavioral) interactions, thus it is inadequate for accommodating such *ad hoc* objects completely.

In this paper, we report an ongoing collaborative project we are engaged in (between HKUST and CSIRO). The project is aimed at extending object-oriented database technology to accommodate non-trivial application dynamics. To this end, we propose a model based on the notion of "conceptual clustering", to facilitate dynamic creation, deletion, and manipulation of *ad hoc* object clusters, which complements existing object class power for accommodating generic application dynamics. Besides the capabilities for objects to be clustered dynamically, we also investigate various implications to the objects that constitute a cluster. While all these are studied generically, a real life application domain is used as a basis to test the ideas. The purpose of this paper is therefore to devise a conceptual model for accommodating dynamic object clusters, and to investigate (practical) solutions to the problems resulting from implementing this model.

The rest of the paper is organized as follows. In section 2 we describe the conceptual model for accommodating various object clusters, and discuss the relationships between clusters and other closely related concepts such as classes, views, and frames. In section 3, we describe a real life database application which is used as a testbed. In section 4, we describe an experimental approach of implementing the cluster model on top of an object database system. Conclusions and future research directions are given in section 5.

2 A Conceptual Model

In this section, we describe a conceptual model for facilitating a useful kind of dynamic function, namely "dynamic clustering" — to dynamically cluster existing objects with dynamically introduced *roles*, forming homogeneous/heterogeneous collections (called "clusters"). There are a number of ways in which clusters can be formed, and in turn this may modify the states and behaviour of the constituent objects (objects that constitute the clusters). Special facilities must therefore be devised in supporting such interactions. In the following, we first establish a taxonomy of clusters necessary for our subsequent discussions, and then define corresponding cluster operators. The relationships and differences between clusters and such related notions as classes and frames are also briefly discussed.

2.1 Clusters: a Taxonomy

We have established a taxonomy of 12 kinds of clusters, based on combinations of three major perspectives: *derivation*, *uncertainty*, and *behavior*. These are described immediately below.

2.1.1 Derivation

There are essentially two ways of clustering from existing object sources: "clustering-by-copy" in forming aggregate-like objects, and "clustering-by-reference" in forming complex clusters. These are captured by the following terms:

- **Deep Cluster**: a deep cluster X is somewhat similar to an aggregate object: it consists of *deep copies* (as defined in SmallTalk-80) of the input objects (taken at the time of the creation of this cluster) as its constituents, with each constituent being assigned a role (i.e., each constituent is a "roled-constituent");[1] as a result of this clustering, the deep copies of the source objects become roled-members *owned* by X (i.e., if X vanishes, so do its roled-members, though the original sources are not effected).

- **Shallow Cluster**: a shallow cluster Y is analogous to an extended complex object, whose constituents are references/pointers (or called *shallow copies* in SmallTalk-80) to the input objects, with each constituent being assigned a role. Y does not own the constituent objects as a result of such clustering (i.e., if Y vanishes, its constituent objects can still exist).

It is also possible for a cluster to be a "hybrid" one, with some of its constituents being pointers to and others being deep copies of existing objects. We omit discussion of such hybrid clusters in this paper, since their features are obvious from the characteristics of a deep and a shallow clusters.

Complementary to the above terms, clusters are further distinguished (with respect to their derivation from sources) to be *adaptive* and *infixed* ones. These are described as follows:

[1] A role here can be an empty one (*void*), a default one, or a newly introduced one; see section 2.2.

- **Adaptive Cluster**: an adaptive cluster X is a cluster whose constituent objects are subject to some "local adaptations" when X is formed. Such adaptations include: to temporarily filter out from its constituents certain properties (non-relevant ones), to add certain presumptive properties, or to assume some hypothetical states (attribute values) to its constituents, all of which are of *local effect* only (i.e., they do not carry any global effect outside of the cluster X).

- **Infixed Cluster**: on the other hand, a cluster may be formed directly from the source objects as its constituents. Specifically, we call a cluster Y an infixed cluster if its constituent objects are directly accepted without any above mentioned "local adaptations" (i.e., none of their previous properties or states will be locally invalidated/suspended in Y, and all changes must be of global effect).

Combining this later aspect to our previous terms, we obtain the following possible forms of clusters:

	Adaptive	Infixed
Deep	DAC	DIC
Shallow	SAC	SIC

2.1.2 Uncertainty

An orthogonal yet related issue on clusters concerns uncertainty of the constituent objects. In particular, a cluster can have component objects that are fully determined, only partially identified, or are identified disjunctively. That is, there may be an issue of certainty/uncertainty involved with a cluster. From this perspective we specify the following:

- **Explicit Cluster**: a cluster X is explicit if every constituent object of X is fully determined at the time when X is formed.

- **Obscure Cluster**: a cluster Y is obscure if Y contains some "vague constituents" (constituents that are not fully determined or identified). An obscure cluster may or may not become explicit later on, depending on the nature and degree of the uncertainty involved.

Applying the above terms to our earlier table, we obtain the following taxonomy:[2]

[2]Note that in the resultant taxonomy, ODIC and ODAC are not implementable if the candidate constituents cannot be identified (hence their deep copies cannot be taken until they are fully instantiated later on).

	DAC	DIC	SAC	SIC
Explicit	EDAC	EDIC	ESAC	ESIC
Obscure	ODAC	ODIC	OSAC	OSIC

2.1.3 Behaviour

Besides the above two important factors, there is yet a third factor pertinent to object clustering, namely, behavioral interactions out of the clustering. Specifically, we are concerned here with whether as a consequence of the clustering, the constituent objects entail new behavior (either explicitly or implicitly) which interacts with their pre-existing behavior and/or with the resultant cluster's. From this perspective we define the following:

- **Loosely-coupled Cluster**: a cluster X is loosely-coupled (or simply L-coupled) if, as a consequence of the clustering, X's source objects are not effected in any way with respect to their properties and states (attribute values). Note that a loosely-coupled cluster is different from an infixed cluster: though an infixed cluster does not have local adaptations, global changes (in terms of properties and states) are still possible to its constituents (through their respective classes), which is not the case in a loosely-coupled cluster.

- **Tightly-coupled Cluster**: a cluster Y is tightly-coupled (or simply T-coupled) if, as a consequence of the clustering, Y's source objects can be effected in terms of their properties and states. This typically involves some *behavior interactions* (through some methods) between Y and its constituents, and/or among the constituents themselves.

Applying above definitions to our earlier taxonomy, we obtain a set of clustering primitives/concepts, which are both semantically and behaviorally rich and expressive. These are described immediately below. Note that in the resulting taxonomy, deep clusters are not compatible with tightly-coupled; rather they are always loosely-coupled with respect to their source (input) objects.

	EDAC	EDIC	ESAC	ESIC
L-coupled	√	√	LESAC	LESIC
T-coupled	X	X	TESAC	TESIC
	ODAC	ODIC	OSAC	OSIC
L-coupled	√	√	LOSAC	LOSIC
T-coupled	X	X	TOSAC	TOSIC

2.2 Operators Over Clusters

To support dynamic cluster creation, deletion, and manipulation, we have identified a collection of basic clustering operators for coping with various "application dynamics" prevailing in real life database applications. The operators are divided into 2 categories: generic cluster operators, and special-purpose operators.

2.2.1 Generic Cluster Operators

Operators of this category are general-purpose: they are applicable to any type of clusters specified above. There are 12 such generic operators:

1. *CLUSTERING(C: cluster-name, DR: default-role, S: set-of-objectref, M: mode): Oid.* This operator forms a cluster object C from a specified set S of input objects which takes DR as their default role within the cluster; DR may be an empty role (*void*), and objects in S may or may not be of the same type (i.e., they can be heterogeneous). The fourth argument determines one of the two modes for the clustering, i.e.:
   (a) clustering-by-copy: when M = *"Deep"*, this operator creates cluster C by taking copies of the objects in S, resulting in a deep cluster which can be inclusive or exclusive, and explicit or obscure (i.e., one of the four possibilities: EDAC, EDIC, ODAC, ODIC);
   (b) clustering-by-reference: when M = *"Shallow"*, this operator creates cluster C by including object references (Oids) in S into C's extent, resulting a shallow cluster which can be one of the other eight possibilities described above.
2. *DECLUSTER(C: cluster-ref).* This operator deletes a cluster denoted by the cluster-reference C (C is either the name of the cluster, or a variable of type Oid holding the cluster's Oid) from the database. If the cluster denoted by C (called cluster C for simplicity) is a deep cluster, all its components and C itself are removed; if C is a shallow cluster, all its constituents (references to the source objects) are excluded from C, and C itself is removed from the database (including all its attributes and methods).
3. *SET-DEFAULT-ROLE(C: cluster-ref, DR: default-role).* This operator (re-)sets the default role of cluster C to DR. All C's constituents/members which take C's default role as their role will be (re-)set consequently.
4. *ADD-TO-CLUSTER(C: cluster-ref, O: objectref[, R : role]).* This operator adds an object O with its specified role R (if given) to the cluster C. A default role prescribed by C will be assigned to O if the third argument is omitted. In the case that the cluster is a deep cluster, a deep copy of O is taken and included to C; otherwise (i.e., C is shallow), O's Oid is added to C's extent.
5. *REMOVE-FROM-CLUSTER(C:cluster-ref, O: objectref).* This operator causes the specified object O and its associated roles to be removed from the cluster C. If C is a deep cluster, the deep copy of O will be deleted from the database completely.
6. *ADD-PROPERTY(C: cluster-ref, T: property-ref).* This operator adds a property T (an attribute or a method) to the cluster C. The property T can be a brand new one, a reference to one of its component object's, or a composition of its components' (e.g., if the property is a method).
7. *DELETE-PROPERTY(C: cluster-ref, T: property-ref).* This operator deletes an existing property T (an attribute or a method) from the cluster C.
8. *ADD-ROLE(C: cluster-ref, O: objectref, R: role).* This operator adds a role R to the specified object O in the cluster C. Note that an object in a cluster can acquire more than one role.
9. *DELETE-ROLE(C: cluster-ref, O: objectref, R: role).* This operator deletes a role R to the specified object O in the cluster C. If R is the only role O has, then O becomes a role-less constituent/member (i.e., O is of *void* role).
10. *ITERATE-OVER-EXTENT(C: cluster-ref[, R : role]): set-of-objectref.* This operator returns a set of (heterogeneous) object references by iterating over the extent of the cluster C. If the second argument R is given, then it returns all the references to those constituents holding the specified role R in C.
11. *STORE-CLUSTER(C: cluster-ref): Oid.* This operator is used to make a cluster C persistent in the database. (Note that clusters by default are temporary/non-persistent.)
12. *REMOVE-CLUSTER(C: cluster-ref).* This operator removes a persistent cluster from the database.

2.2.2 Special-Purpose Operators

Operators of this category are generally only applicable to certain types of clusters but not all. There are 10 operators in this category, where the first three operators are defined for obscure clusters, the next three specifically for adaptive clusters, and the last four are applicable to all tightly-coupled clusters.

1. *CREATE-VC(D: domain, P: condition): Oid.* This operator generates a virtual constituent (VC) holding the condition P on the specified domain D; it is used to form an obscure cluster temporarily. D can be a class, a cluster, a result of some (intermediate) query, or combinations of these. P is either a declarative predicate or a procedural definition. An important example of a VC (and an obscure cluster) is given in section 3.2. The VC can be subsequently instantiated/bounded or removed by the next two operators. An instantiation may bind a VC to a simple object, or a set of objects, as determined by its nature (i.e., its condition P).

2. *INSTANTIATE-VC(V: VC-ref, M: method): Oid.* This operator instantiates a VC according to a specified method M, causing the VC to be bound to a (set of) fully identified object(s); the VC itself vanishes if its associated condition has been completely satisfied.

3. *REMOVE-VC(V: VC-ref).* This operator removes a VC without instantiating it.

4. *FILTER-OUT(C: cluster-ref, SP: set-of-properties).* This operator is used to form an adaptive cluster C by filtering out irrelevant or non-applicable properties SP (attributes and methods) from its constituent objects. The properties in SP can be original ones and/or presumptive ones added by the next operator.

5. *ADD-PRESUMPTIVE-PROPERTY(C: cluster-ref, P: property-ref$[, R : role]$).* This operator causes a new presumptive property P (attribute/method) to be temporarily added to all C's constituents holding (the default) role R (if specified);[3] if R is not specified, then P is added to all C's constituents indiscriminately. The effects of this operator (and the next associated one) are only of local scope inside C.

6. *SET-PRESUMPTIVE-VALUE(C: cluster-ref, A: attribute-ref$[, R : role]$, V: value).* This operator sets a presumptive value V to a specified attribute A of all C's constituents holding (the default) role R (if R is specified). If R is not specified, but A is fully specified (with form O.A where O is a constituent object in C), then the presumptive value V is assigned to O.A only; otherwise V is assigned to all C's constituents having A indiscriminately.

7. *IMPOSE-PROPERTY(C: cluster-ref, P: property-ref$[, R : role]$).* This operator is similar to the ADD-PRESUMPTIVE-PROPERTY operator, except that it is applicable to tightly-coupled clusters, in the sense that the new property P to be imposed is of global effect (and possibly permanent effect if C is made to be a permanent cluster) outside of C.

8. *SUSPEND-PROPERTY(C: cluster-ref, P: property-ref$[, R : role]$).* This operator causes the specified property P (attribute/method) of C's constituent objects (holding role R if R is given) to become suspended globally. Clearly, the effect of this operation can also be permanent if C is a persistent cluster.

9. *RESUME-PROPERTY(C: cluster-ref, M: method-ref$[, R : role]$).* This operator causes a suspended property of C's constituent objects (holding role R if R is specified) to become valid again globally. Note that this operation is always of permanent effect, no matter if C is permanent or temporary.

10. *INVOKE-METHOD(C: cluster-ref, P: method-ref$[, R : role]$, A: list-of-arguments).* This operator invokes a specified method P using given parameters in A. The method P can be a method of some constituent objects (holding role R if R is given) in C, or a method defined for C itself.

[3]Note that if the third argument R is given, P can be viewed as an associated property of the role R in the context of the cluster C.

2.3 Discussions and Comparisons

So far we have established a taxonomy of 12 kinds of clusters, based on the applicable combinations of **derivation**, **uncertainty**, and **behaviour** perspectives. We also defined associated cluster operators in supporting the 12 kinds of clusters and their capabilities. From the above descriptions, we see that there exists a resemblance between a cluster and a class. Like a class, a cluster has a collection of objects in its extent (as its constituents/members), and associated operations such as iterators over the constituents/members of its extent (as described above). But it differs from a class in several fundamental aspects: (i) it is an extended aggregate/composite object consisting of a collection of (typically heterogeneous) objects,[4] rather than a warehouse holding a set of uniform objects; (ii) it supports the notion of dynamic "roles" [LPS91, Per90], and is a dynamic construct useful for transient, tentative, and/or irregular situations, whereas a class is a statically defined construct for stable/regular sets of objects (with frames for prototypical situations); (iii) it only includes/removes existing objects to itself, and does not create "new" objects (except for new roles and the cluster itself); (iv) it allows individuality of its constituent/member objects to be expressed (partially through the roles), whereas a class emphasizes the commonalities of its objects; (v) it supports behavioral interactions between the cluster and its constituent/member objects (and possibly among the constituents themselves) — another feature neither supported by a class nor existing in a frame.

Recently, there have been some proposals for view support in object databases (e.g., [AB91, SLT91]). Compared with classes, views are closer to our clusters because views basically are also derived from existing objects. However, since views are just another kind of class, most of the above points on comparing clusters and classes are still applicable to views, except for point (iii) as views also do not create new objects. The work on updatable views in [SLT91] would also veto point (v) partially but not completely, as views typically do not facilitate behavioural interactions among their objects.

3 Real Life Examples

To demonstrate the validity and usefulness of conceptual clusters, we introduce an example here in which clusters are found to be a natural and suitable knowledge representation constructs. These real life examples are drawn from a decision support system application. A decision support system provides a rich application environment for a wide range of information technology as shown in Figure 1. In the scope of this paper we cannot even attempt to summarize the modules of a decision support system. However the reader can appreciate that these systems aim to provide a user with assistance in complex problem solving. Typically clusters are invoked in a problem solving model used in conjunction with a knowledge/data base (i.e., an information base).

[4]From this viewpoint, a cluster is closer to a frame object; however, as shown in subsequent discussion, it also differs from a frame in other aspects.

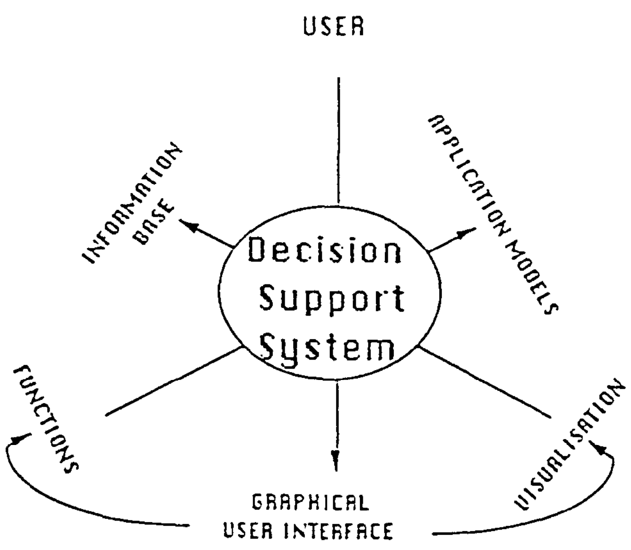

Figure 1: A Decision Support Information System

3.1 The Inquiry Problem

Frequently today judicial bodies are set up to inquire into alleged serious corrupt behaviour by individuals in the community. The inquiry body has certain powers invested in it to subpoena witnesses and take evidence. Its goal is to present a report which will contain conclusions and recommendations about the situation.

A simple information model of the process is summarized below.

Event (e): A well documented factual occurrence in the past.

Witness (w): An individual who makes statements to the inquiry.

Statement (s): An assertion made by a witness about a player or players.

Player (p): An object involved in the inquiry

Conclusion (c): An interim or final assertion made by the inquiry body.

An object-oriented analysis and design might proceed to identify classes and objects, structures, attributes, services etc. by elaborating on the above model. However conclusions would pose a problem for this analysis, because they are extremely variable in behaviour and properties. In the dynamic processes of this decision support application, conclusions would be subject to revision, refutation, coexistence with conflicting conclusions, etc. In addition the formation of a conclusion would involve collecting together numerous objects from the database in their respective roles (viz. clustering).

There are clearly many overtones of artificial intelligence problems in this example (eg default reasoning, non-monotonic reasoning, uncertainty reasoning, constraint based reasoning, truth maintenance etc.). We are interested here in extending the capability of object-oriented technology to provide a more flexible and efficient representation capability. We now illustrate the use of clusters in this dynamic application.

3.2 Clusters in the Inquiry Example

The simplest conclusion is formed as a result of a single statement. For example suppose there was an statement s1 describing an action by player p1. A conclusion c1 could be formed about p1 from the details of s1. Clearly the conclusion would be dependent on s1. This is represented by a cluster having two constituents, p1 with role '`about`', and s1 with role '`supporting-evidence`'.

It is possible to anticipate certain roles for constituents, and the above example could readily be accommodated in a predefined class with the roles as client relationships. However as the inquiry process evolves the number of roles will grow and defeat the class definition approach.

For example, suppose we have the following objects:
e1 : Region r1 was rezoned from classification A to classification B on date d1
e2 : Player p1 purchased land in r1 on date d2 (d2<d1)
s1 : Player p1 knew player p2 before date d2
s2 : Player p2 obtained information on date d3 (d3<d2) about the proposed rezoning of r1
s3 : Player p1 was introduced to player p2 after date d1

It may be useful to form a conclusion cluster:

c1 := { **`confidence`**: probable

`corrupt-transaction`: p1, p2

`related-events`: e1,e2

`evidence-for`: s1,s2

`evidence-against`: s3

`possible-witness`: w7,w9

`others`: ...

}

The role '`possible-witness`' in the conclusion has less generality than the others, and suggests that many such roles would not be realised until the application was underway.

Shallow clusters would be the automatic choice in this example. Hybrid clusters may arise, for example where a statement is adapted for the purpose of a conclusion.

We can extend the above example with the following information leading to an associated conclusion with an obscure constituent.

s4 : Player p1 knew player p3 before date d2

s5 : Player p3 obtained information on date d3 (d3<d2) about the proposed rezoning of r1

s6 : Player p1 was introduced to player p3 after date d1

The associated conclusion with an obscure constituent is:

c2 := { **`confidence`**: most likely

`corrupt-transaction`: p1, p2 $\vee$ p3

...

}

Subsequently on the basis of statements s57, s58 made by witnesses w7 and w9 we may replace the conclusions c1 and c2 with

c24 := { **`confidence`**: confirmed

`confirmed-conclusion`: c1

`replaced-conclusions`: c1, c2

`conflicts`: s17, c18

`discussed-with`: legal3

`major-evidence`: s1, s57

`minor-evidence`: s2, s58

...

}

Clearly, tight coupling arises in the case of the `confirmed-conclusion` role of c1, and possibly in the case of the **`conflict`** role of s17 and c18 as well. On the latter, a message which performs verification on statement objects (e.g., recalls a witness) may result, and the previous unconfirmed conclusion c18 may be refuted.

3.3 Conclusion Cluster Services

A number of generic services are desirable for conclusion clusters. For decision support the most important are search and visualisation of the information, based on:

- query capability which allows role qualification
- instantiation of the predicates of obscure constituents
- navigation of objects by role relationship
- inverse queries from constituent qualification

Our cluster model provides a very suitable basis upon which such generic services can be provided. Clearly role names here capture much of the semantics of the application, thus explicit support of roles is not only desirable but also important. Furthermore, it also provides us with such useful "by-products" as to be able to model multi-faceted objects [Sci89].

4 An Experimental Implementation

To demonstrate the feasibility of our clustering model, a prototype implementation has been designed and implemented. The prototype system is implemented in a persistent C++ environment on Sun4, utilizing an object-oriented database development system, namely ONTOS. In this section we specify our design and implementation approach in such a comprehensive software engineering environment. First, we briefly introduce certain aspects of the ONTOS system, which are necessary for our subsequent discussions. An implementation approach, based on the notion of "meta-cluster", is then described, utilizing existing ONTOS class facilities. Such a class-based implementation suggests that a cluster can also be viewed as (fundamentally) an instance of some "extended" abstract data type.

4.1 The ONTOS Environment

ONTOS is an object-oriented database programming system developed by Ontos, Inc. [Ont91]. It uses an object model internally, thus directly expressing the complex relationships between data, and provides aggregate classes for modeling one to many relationships. A major application program interface it supports is a C++ interface, along with an SQL interface for programmatic queries [AHS91]. While there are many other aspects and features (such as ONTOS transaction mechanism, version control, server-client architecture, exception handling, etc.), we restrict ourselves here to the most directly relevant aspects — ONTOS C++ interface and class library.

4.1.1 C++ Interface and Class Library

The ONTOS C++ interface consists of a relative handful of meta classes linked into the user's application and a small class library. It resolves references directly and presents objects that are fully compatible with the C++ language [Str86]. Its class library provides classes of Aggregates and Iterators as well as classes for schema definition. In particular, the ONTOS class library introduces an `Object` class, which is the parent of all *persistent* classes. `Object` defines a constructor for creating objects in the database and a destructor for deleting them. It also defines properties for an object name (hence object can be referenced both by name and by reference) and for the "physical clustering" of objects in the database to achieve greater performance. The library also contains schema classes to represent class definitions (e.g., properties, member functions, arguments etc.) in a run-time accessible way. Furthermore, Classify, an ONTOS utility, is provided to generate schema objects from standard C++ class definitions. These objects are loaded into the database to represent the database schema and are accessible both to ONTOS and to the application.

ONTOS 2.1, the latest release from Ontos Inc., supports the C++ 2.0 model, in which multiple inheritance is allowed (i.e., a child class can have more than one immediate parent classes). This feature exhibits a more powerful capability in modeling shared subclasses, and allows more concise representations and better re-usability of class definitions. It will therefore be utilized in our prototype implementation (see below).

4.2 A Meta-Cluster Approach

The power and flexibility of ONTOS and its associated C++ interface provides a suitable implementation base for the proposed dynamic clustering model. As a cluster is conceptually different from a class (note that a cluster is itself a complex object), an intuitive mapping and extension from a class to a cluster will not work. Rather, a cluster is defined as an instance of an "extended" abstract data type (the meta-cluster `Cluster`), with a meta-cluster lattice hierarchy being introduced at an abstract level (as described below).

4.2.1 The Built-in Cluster Hierarchy

Figure 2 illustrates a meta-cluster lattice that we introduce into the ONTOS existing class mechanism. This newly built-in lattice (hierarchy) provides the basic dynamic functions required by various clusters, which are not supported by a conventional class hierarchy. The lattice is rooted at `Cluster` which is defined immediately under the node `Object` of ONTOS, since certain clusters may be desired to be made persistent under certain circumstances (cf. section 2.2.1).

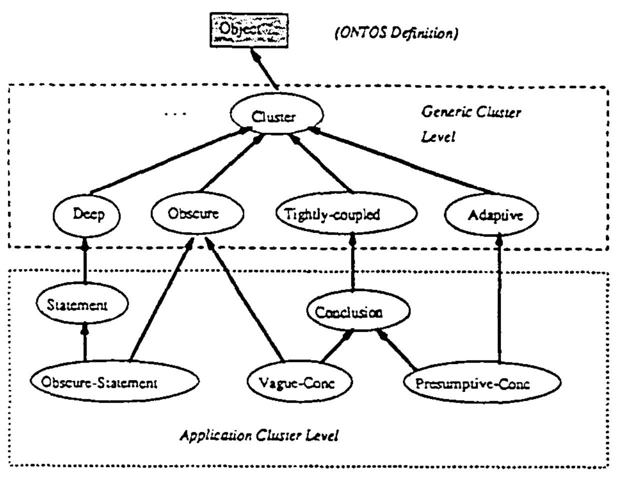

Figure 2: The Built-in Cluster Lattice inside ONTOS

4.2.2 Design Decisions and Implementation Issues

As shown in Figure 2, this newly added lattice introduces several abstract data types (ADTs), which support various forms and characteristics of the aforementioned clusters. In designing the prototype, we have made the following decisions in addressing specific implementation issues:

Default clusters In the established cluster taxonomy, there are altogether 12 possible forms of cluster. Since some of them are formed by combining "seed" clusters (e.g., EDIC from `explicit`, `deep`, and `infixed` combination), it is not necessary to support such *derived* ones explicitly. Furthermore, as clusters in real life applications tend to exhibit certain "default" forms, we can reduce the number of ADTs by imposing default manners for clusters. In particular, in our prototype system clusters are defined as `shallow`, `infixed`, `explicit`, and `loosely-coupled` by default. This is realized by implementing the meta-cluster `Cluster` (which is the root of this newly introduced hierarchy) to support all the general clusters functions (cf. section 2.2.1), plus the ones applicable to these four types of clusters (cf. section 2.2.2). The default manner may of course be overwritten by a cluster if the cluster is created directly at a lower level.

Data structure for clusters Fundamental to the implementation of the cluster model is a "right" data structure for clusters. As a cluster is essentially a variable collection of `<object, role>` pairs, an immediate choice is to use Class List (or Class Array) in C++ [Str86]. In this approach, a cluster is implemented as a list containing a (variable) number of instances of some *ad hoc* class (say, a class called `Constituent` which captures the `<Oid, role>` relationships). Note that this approach can be easily extended to support other types of clusters (e.g., roleless clusters, clusters of primitive data objects, etc.), through the so-called parameterized type or template approach [Str88].

Local effect of adaptive clusters While using the right data structure allows us to implement most of the forms of clusters with relative ease, certain kinds of cluster require special treatment and techniques. Among them, adaptive clusters are challenging ones to implement on a conventional object-oriented system such as C++. As described in section 2, an adaptive cluster has a special characteristic, namely its local adaptations (i.e., changes to its constituents' definitions and/or states inside the cluster). While for deep clusters this is not a problem, it becomes a non-trivial one for shallow clusters since in a shallow cluster, its constituents are supposedly taken directly from their original classes. Therefore we are facing a somewhat paradoxical situation: on one hand, we need to guarantee the local effect of the local adaptations to the constituents, on the other hand, we need to allow "applicable" global changes (those not being locally filtered and adaptated) still to take place on those constituents.

To adequately implement shallow adaptive clusters, a technique called *Oid-forwarding* is used for "simulating" the desired local effect. Informally, a constituent object Oj in a shallow adaptive cluster is simulated by creating a shadow copy[5] (say $\hat{Oj}$) of the source object (i.e., Oj), and by attaching a *.forward* entry (similar to the *.forward* file in Unix mailing system) to Oj, inside which multiple destinations (e.g., $\hat{Oj}$ and Oj) can be specified. This *.forward* entry will assure that all subsequent messages to Oj will be forwarded to the specified destinations (i.e., Oj and $\hat{Oj}$). In this way, applicable global changes can still be propagated to Oj's shadow object $\hat{Oj}$, with the other direction being not possible (i.e., changes to $\hat{Oj}$ will only be of local scope). Figure 3 illustrates this scenario of Oid-forwarding.

Behaviour interactions of tightly-coupled clusters Another type of clusters which requires non-trivial implementation is tightly-coupled cluster, due to the possible behaviour interactions. Within a

[5]A shadow copy is essentially a deep copy, with additional capability of reflecting "applicable" changes from external sources, as described below.

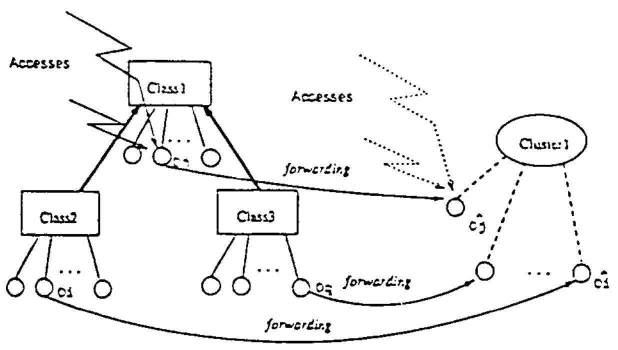

Figure 3: Oid-Forwarding for Implementing Shallow Adaptive Clusters

tightly-coupled cluster, a constituent object may have four kinds of behaviour (method) distinguished:

1. **Newly introduced methods** that do not conflict with existing ones. This kind of behaviour is straightforward to implement in C++, based on its existing method mechanism.

2. **Dominant methods** that replace existing ones. This type of methods correspond to those that have the same names with existing ones, but with modified procedure bodies. Messages sent to the constituent are to be directed to the dominant methods which replace existing ones. If the cluster is made persistent, then those replaced methods are essentially *relinquished* forever (provided that their corresponding dominant ones are not dropped). If later on a dominant method is dropped (e.g., when its associated role is removed) from the constituent, the corresponding method will become effective again. Due to these characteristics, a more sophisticated "virtual function" mechanism than that offered by C++ is required. In particular, the virtual function mechanism needs to be extended with the capability of assigning different weight to its active definitions (methods), and (re-)direct a message to the one with the highest weight. Such an extended mechanism is also desired by the next two types of behaviour interactions.

3. **Suspended methods** correspond to those that are temporarily "turned-off" (through explicit suspension). This can be viewed as a special case of the previous one if we regard suspended methods being temporarily replaced by empty ones (i.e., their dominant methods are empty procedures). Therefore we can handle this type of behaviour interaction through the same meachanism described above.

4. **Resumed methods** that were suspended before and become re-usable (through explicit resumption). This again can be viewed as a special case of the second case, in the sense that the resumed methods are essentially replaced methods with their dominant ones being dropped from the constituent. Therefore its implementation becomes straightforward with the afore-mentioned mechanism.

Instantiation procedures for obscure clusters The final type of clusters which is not readily implementable based on current object-oriented capabilities is obscure cluster. As defined in section 2, an obscure cluster is the one which contains "vague constituents" (e.g., in the form of partial references or disjunctive references) that are to be instantiated. This type of cluster is typical in hypothetical reasoning applications [LS91]. Since logic reasoning capabilities are not supported in current object-oriented languages and systems, automated instantiation procedures based on reasoning are not implementable without integrating the object system with an AI system (ruled-based or frame-based one). For the current prototype, such instantiation procedures will therefore be simply interactive ones (viz., user's responsibility). A possible candidate for providing such reasoning capabilities into Ontos is CLIPS - a C Language Integrated Production System [Cli89].

4.3 Discussions

In this section we have described a straightforward implementation approach of our cluster model, based on the ONTOS object model and class facilities. In particular, from an implementation level a cluster is treated as a complex object with additional dynamic functions (viz. dynamic updates of the states and behavior of input (component) objects, dynamic instantiation/binding of virtual components, and dynamic filtering capabilities). While such a class-based implementation approach allows us to come up with a quick-and-dirty prototype by utilizing to the extent possible existing ONTOS facilities, it is not the "best" approach for implementing clusters. Indeed, a more radical and efficient approach would be to build the cluster mechanism from scratch, which would put the `Cluster` at the same level as ONTOS `Object`, with a common parent (e.g., an `Entity`) being introduced for supporting persistence, identity, iterators, and inheritance, etc. One obvious advantage of the latter approach would be the support of a more powerful class mechanism (by defining a shared subclass from these two), supporting powerful features from both sides (e.g., uniformity, dynamic functions and roles, and so on). We are currently investigating this approach.

5 Conclusions and Directions

The Conceptual Clustering Model (CCM) presented in this paper aims at extending current object-oriented database technology to accommodate important application dynamics, through facilitating dynamic creation, deletion, and direct manipulation of *ad hoc* object clusters. Such dynamic clusters complement existing object classes of an object-oriented database in modeling irregular, tentative, evolving, and multi-perspective objects, allowing dynamic inter-object relationships and behaviour to be captured on the fly (through their dynamic roles and associated methods). A taxonomy of 12 kinds of clusters has been established from the perspectives of derivation, uncertainty, and behaviour interaction (and their applicable combinations). An AI reasoning application has been used as a testbed for this comprehensive model, and an implementation approach based on a persistent C++ object database system has been described.

From model refinement and enrichment point of view, further important issues such as cluster role manipulations, inter-cluster interactions and derivations (e.g., to copy or derive clusters from existing ones), and dynamic constituent instantiations need to be addressed. On the latter, we are exploring the suitability of incorporating a rule base (the afore-mentioned CLIPS) into the database system (i.e., ONTOS), in hope of acquiring the desired reasoning capability. Our continuing prototype implementation experiment should also provide significant feedback on refining the cluster model and operations (from the practical and feasibility viewpoint).

Finally, our CCM model described in this paper is being further investigated and applied to a larger scale real life reasoning application. The goal is to provide an advanced object-oriented environment inside a decision support system, in which the information units can change in real time. Our cluster model provides a useful basis upon which important services to this type of applications can be devised and supported (cf. section 3.3). Other interesting applications we plan to look into include using clusters as a means for knowledge synthesis and discovery, in order to support database integration, object migration, and information sharing in a distributed, multiple databases environment.

ACKNOWLEDGEMENT

The authors wish to thank William Sun for helping constructing the prototype system, and Peter Milne for his constructive comments and proof-readings on an earlier version of the manuscript.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 238–247. ACM SIGMOD, 1991.

[AHS91] T. Andrews, C. Harris, and K. Sinkel. Ontos: A persistent database for c++. In R. Gupta and E. Horowitz, editors, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, pages 387–406. Prentice-Hall, 1991.

[Cli89] NASA Johnson Space Center (AI Section). *CLIPS Reference Manual - Version 4.3 of CLIPS*, 1989.

[KLE89] W. Kim and F.H. Lochovsky (Editors). *Object-Oriented Concepts, Databases, and Applications.* ACM Press, 1989.

[LPS91] Q. Li, M. Papazoglou, and J. Smith. Dynamic object models with spatial application. In *Proceedings of the 15th Int'l Computer Software and Applications Conference.* IEEE Computer Society, Tokyo, Japan, September 1991.

[LS91] Q. Li and J. Smith. Dynamic object clustering: an application and an implementation. In *Proceedings of the IJCAI-91 Workshop on Integrating Artificial Intelligence and Databases.* IJCAI, Sydney, Australia, August 1991.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[Ont91] Ontos Inc., Three Burlington Woods, Burlington, MA01803. *ONTOS Reference Manual - ONTOS Release 2.1*, 1991.

[Per90] B. Pernici. Objects with roles. In *Conferences on Office Information Systems*, pages 205–215. ACM, 1990.

[Sci89] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.

[SLT91] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *The 2nd Int'l Conference on Deductive, Object-Oriented Databases*, Munich, Dec. 1991.

[SN88] M. Schrefl and E. Neuhold. Object class definition by generalization using upward inheritance. In *IEEE Database Engineering*, pages 4–13. IEEE, 1988.

[Str86] B.L. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

[Str88] B.L. Stroustrup. Parameterized types for c++. In *Proceedings of the Usenix C++ Conference*. Denver, 1988.

[ZME90] S. Zdonik and D. Maier (Editors). *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.