Towards a Comprehensive Concurrency Control Mechanism for Object-Oriented Databases

David H. Olsen
University of Akron

Sudha Ram
University of Arizona

Object-Oriented databases are becoming increasingly popular in business. Issues such as query optimization, analysis and design techniques, and concurrency control have been addressed as they pertain to the relational model but have not been addressed as they apply to the object-oriented model. This paper includes the framework development and description of a concurrency control mechanism named O^2C^2 which is specifically designed for an object-oriented database. O^2C^2 is a lock–based concurrency control mechanism that forms the basis of this research.

A description of database concurrency control and object-oriented database precepts are presented to provide a basis for a comprehensive framework for concurrency control in object-oriented databases. The theory is developed along four specified dimensions which are the hierarchical level dimension, the data type dimension, the composite or complex objects dimension and transaction type dimension. Additionally, a comprehensive list of rules is given that are crucial to an object-oriented database concurrency control mechanism. The rules are given to provide a basis not only for the O^2C^2 mechanism, but for any object-oriented database concurrency control mechanism. The O^2C^2 mechanism is then presented after which a discussion ensues about the possible transaction types in order to demonstrate the robustness of the mechanism.

Concurrency control (CC) is defined by Bernstein et al. (1987) as follows: “concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other.” In other words, CC mechanisms interleave the operations of competing processes in such a way that consistency is maintained. It is also one of a few critical components of a database management system. A few basic approaches to concurrency control have been proposed and developed in the database area. Based on these approaches, several hundred of algorithms have been created (Barghouti & Kaiser, 1991; Bernstein, et al., 1987). However, very few of these address object–oriented databases.

The concurrency control mechanism O^2C^2, described in this research contributes in two important ways. First, the concurrency control mechanism O^2C^2 is presented along with how it relates to the developed framework. O^2C^2 is lock–based and two–phased but is more complex than standard algorithms in order to deal with the complexities of object–oriented databases. Second, it adds to the body of theory that explains the objectives, problems, and tradeoffs in the area of concurrency control for object–oriented databases by comparing it to the mechanisms in ORION and O_2.

The advantages of O^2C^2 include a higher degree of concurrency than current implementations. It is superior to other approaches because more transactions are allowed to process at the same time due to fewer conflicting lock types and a finer granularity of lock types. In a majority of circumstances, the higher degree of concurrency yielded by O^2C^2 results in higher throughput which translates into better per-

Manuscript originally submitted July 18, 1994; Revised January 18, 1995; Accepted April 12, 1995 for publication.

formance.

Concurrency control is critical because it is the deciding factor of database performance. In Franaszek, et al. (1992) it is shown that hardware advances have led to several fold improvements in performance and that if the trend continues, CC algorithms will have to be much more efficient just to keep pace with hardware. Indeed, data contention is increasingly the single most critical factor of database performance.

Concurrency control in an object–oriented database is necessarily more complex than it is in other types of databases due to several reasons such as inheritance, the fact that updates to certain parts of the database are not independent from updates to other parts, and the nature of what constitutes an object.

High performance CC mechanisms that produce correct schedules are crucial to the success of any database (Bernstein, et al., 1987). Indeed, any database model will exist only in theory until issues such as concurrency control are examined. The object–oriented database field is emerging as a critical area in database research but work in concurrency control has so far been limited to versioning (Cattell, 1994) and some simple hierarchical locking matrices. A comprehensive theory does not yet exist explaining the objectives, problems, and tradeoffs that must be examined in the area of concurrency control for object–oriented databases.

The paper proceeds with section 2 providing an introduction to concurrency control and object–oriented databases. Section 3 describes the disadvantages and problems of current approaches to object–oriented database concurrency control. Section 4 provides a framework for object–oriented database concurrency control while section 5 includes a description of O^2C^2 detailing the types of locks and when they are used. Section 6 is a summary of the performance analysis described in other research and section 7 concludes with a discussion of the implications of this future research.

Concurrency Control and Object-Oriented Databases

Concurrency Control Mechanisms

Along with coordinating transaction processing in a manner that maintains consistency, it is implicit in concurrency control studies that performance is the other critical factor. That is, all other things equal, mechanisms that support higher levels of performance for a given task are more desirable than mechanisms that support lower levels of performance. In reality, each of the hundreds of CC algorithms provide tradeoffs depending on the characteristics of the application (Bernstein, et al., 1987; Gray & Reuter, 1993; Papadimitriou, 1979). For the purposes of this paper then, concurrency control will be defined as follows:

A concurrency control mechanism allows multiple users to access and update the database so that overall correctness is maintained and performance is optimized. This means that each transaction is executed as though it were processed in isolation (in order to maintain consistency), yet, throughput is maximized.

In the database area, the three standard CC implementations are two–phase locking, timestamping, and optimistic protocols (Bernstein, et al., 1987; Eswaran, et al., 1976; Gray, et al., 1976;Kung & Robinson, 1981). The three CC approaches are briefly reviewed in the following paragraphs. Locking however, was chosen in this research study for two reasons. First, it is the approach most often used for database implementations because it is well understood and can be used for general transactions (Bernstein, et al., 1987; Eswaran, et al., 1976; Gray, et al., 1976). Second, locking schemes are described for a few current object–oriented database implementations including ORION and O_2. These lock–based schemes provide the basis of comparison in a performance study.

Locks are commonly used in order to create a schedule that results in a consistent database. Consistency is determined by comparing the schedules a CC mechanism produces with serial schedules: that is, schedules that only allow one transaction at a time to execute. Serializability is the standard of consistency by which all CC mechanisms are compared. Some basic rules for two–phased locking include (Eswaran, et al., 1976):

1. Whenever a transaction reads or writes a data item, it must hold some kind of lock on that item.
2. At some point before the transaction finishes and becomes inactive, all locks must be released.
3. For any transaction, all lock requests must precede all unlock requests. This means an unlock request is never followed by lock request. If this happens, the transaction must be forced to abort. This is the heart of two–phased locking.
4. A transaction cannot write into the database until after the commit point is reached. This is done to avoid rollbacks (also known as the cascading effect).

These are just the basic rules of two–phased locking. The two–phased protocol is so named because two phases are observed. First, the growing phase accumulates lock requests; second, the shrinking phase unlocks items after processing. Finally, after the transaction reaches the commit point, it is physically written into the database.

The timestamping technique was originally designed for distributed database systems, but many centralized CC mechanisms also utilize some version of timestamps (Bernstein, et al., 1987). The basic rule of timestamp ordering is that older transactions are processed before younger ones. If a younger transaction has already processed a given item and

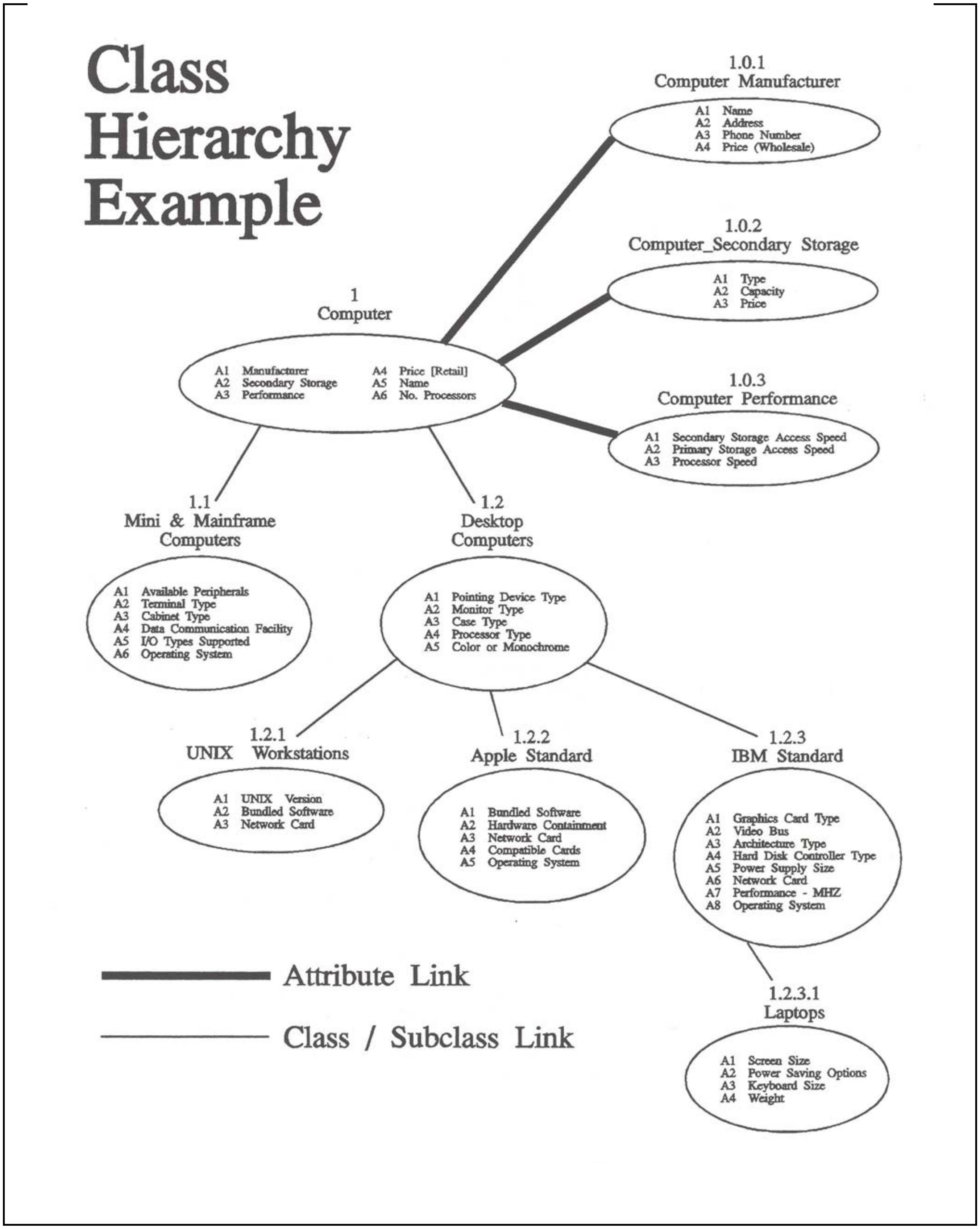


Figure 1: Class Hierarchy Example

CLASS HIERARCHY EXAMPLE

A = Attribute
I = Instance
M = Method
No. = Class Number (Italicized means Attribute Class)

1 Computer
- **A1** – Manufacturer
- **A2** – Secondary Storage
- **A3** – Price
- **A4** – Performance
- **A5** – Name
- **A6** – No. Processors
- **I1** – Cray Supercomputer K
- **I2** – XYZ Corp. Dedicated Database Computer
- **I3** – ABC Corp. Front–End Server
- **M1** – Add an instance of this class
- **M2** – Delete instance from the database
- **M3** – Modify an instance of this class

M4 – Modify Attribute A_1 of this class
.
.
M_n – Modify Attribute A_i of this class

Computer_Manufacturer
- **A1** – Name
- **A2** – Address
- **A3** – Phone Number
- **I1** – IBM
- **I2** – Digital
- **I3** – Apple

Computer_Secondary Storage
- **A1** – Type
- **A2** – Capacity
- **A3** – Price
- **I1** – Fujitsu Model 1020
- **I2** – Seagate 20205
- **I3** – Maxtor Model ABC

Computer_Performance
- **A1** – Secondary Storage Access Speed
- **A2** – Primary Storage Access Speed
- **A3** – Processor Speed
- **I1** – COMPAQ Model A
- **I2** – Apple Laptop V
- **I3** – Digital Mainframe

1.1 Mini & Mainframe Computers
- **A1** – Available Peripherals
- **A2** – Terminal type
- **A3** – Cabinet Type (Big / Huge)
- **A4** – Data Communication Facility
- **A5** – I/O Types Supported
- **A6** – Operating System
- **I1** – IBM System 3095
- **I2** – UNISYS System 1001
- **I3** – HP 3001

1.2 Desktop Computers
- **A1** – Pointing Device Type
- **A2** – Monitor Type
- **A3** – Case Type (Desktop / Tower)
- **A4** – Processor Type
- **A5** – Color or Monochrome monitor
- **I1** – Commodore Model 1
- **I2** – Acari Model 2
- **I3** – Amigo Model 3

1.2.1 UNIX Workstations
- **A1** –UNIX Version
- **A2** –Bundled Software
- **A3** –Network Card
- **I1** – VAX Workstation I
- **I2** –HP Apollo I
- **I3** –SUN SPARC I

1.2.2 Apple Standard
- **A1** –Bundled Software
- **A2** –Hardware Containment
- **A3** –Network Card
- **A4** –Compatible Cards
- **A5** –Operating System
- **I1** – Apple Workstation 1
- **I2** – Apple Educational
- **I3** – Apple Laptop 1

1.2.3 IBM Standard
- **A1** –Graphics Card Type
- **A2** –Video Bus (Local or Standard)
- **A3** –Architecture Type (ISA or EISA)
- **A4** –Hard Disk Controller Type
- **A5** –Power Supply Size
- **A6** –Network Card
- **A7** –Performance (MHZ)
- **A8** Operating System
- **I1** – COMPAQ Model A
- **I2** – Northgate Model Q
- **I3** –Gateway 2000 Model Z

1.2.3.1 Laptops
- **A1** –ScreenSize
- **A2** –Power Saving Options
- **A3** –Keyboard Size
- **A4** –Weight
- **I1** – ZEOS Model 1
- **I2** – Toshiba Model 4
- **I3** – NEC Model 8

updated the database, then the older conflicting transaction is aborted and restarted with a more recent or larger timestamp.

The philosophy behind the optimistic techniques for CC is that in certain systems, conflicting transactions are rare. In systems where conflicts between transactions are rare, locking approaches impose unnecessary overhead and thus restrict the concurrency degree. Throughout their paper, Kung and Robinson (1981) take the optimistic view whenever decisions regarding conflicts must be made but they always allow for the *rare* case of conflicts. They allow all transactions to proceed and then perform avalidation phase immediately prior to the write or commit phase. Kung and Robinson describe optimistic techniques in phases and are known as the read phase, the validation phase, and the write phase.

Object–Oriented Database Management Systems

Object–oriented databases have not only provided a fertile area of research for academics, they are also starting to fill a void in fields where conventional database technology is deficient. Some deficiencies of conventional databases include an inability to manage complex types of data and more realistically model real world data. Areas such as CAD and software engineering are benefitting from the object–oriented database model because it is able to efficiently manage more complex data(Cart & Ferrie, 1992; Cattell, 1994; Garza & Kim, 1988).

Unlike the relational database model, however, there is neither an underlying elegant mathematical model, nor is there a well established, accepted set of concepts. There also seems to be a marketing twist to both research and practice in that many papers and products are incorrectly labeled "object–oriented." Because it is such a promising field, many systems and research projects are touted as "object–oriented"; unfortunately, many fall short of expectations. Additionally, it is interesting to note that many of the concepts of the object–oriented database management system field seem correlated to concepts native to semantic modeling (Hammer & McLeod, 1981; Smith & Smith, 1977).

Kim (Garza & Kim, 1988; Kim, 1990)has stated many basic, important concepts concerning object–oriented databases as a result of the development he led during the creation of the ORION system. It was one of the first commercial object–oriented databases and is considered quite advanced. It is noted, though, that like the relational database area, very few object–oriented database implementations conform to all of the basic theory. For example, most object–oriented databases do not support true multiple inheritance even though the object–oriented database model includes it.

Figure 1 is an example that demonstrates some of the basic concepts in object–oriented databases.

Objects are simply any real–world entities that a user finds valuable to identify. Objects are related to the instance concept in semantic data modeling. An instance object belongs to one class and is an instance of the class in which it belongs. An object can be a member of only a single class. It is possible, however that a class could inherit its attributes and methods from more than one super class. Though multiple inheritance introduces complexity, there are many situations where it would facilitate representation of a given state as well as reduce the amount of development time.

In Figure 1, an example of a class hierarchy is provided. An example of an instance object in this Figure would be a Northgate Model Q personal computer. That object would be an instance of the IBM standard class and would inherit attributes from the desktop computer class and the computer class.

In an object–oriented database management system (OODBMS), each object is assigned a unique ID number. The ID number is theoretically never assigned to another object even if the original object is deleted from the system(Garza & Kim, 1988; Kim, 1990). An instance object consists of values for the attributes of the object. An object that represents the Model Q computer as an instance would have values for the attributes called {Manufacturer}, {Performance}, {Secondary Storage},{Price retail}, {Name}, {No. Processors}, and all attributes that are included in the desktop computer class. In general, an object can be an instance, a class definition, or a method and it is generally advisable to specify the object type for clarification.

The class concept is another important idea that has its roots in semantic modeling (Hammer & McLeod, 1981; Smith & Smith, 1977). A class collects a group of logically similar objects with a defined set of attributes and all objects that are an instance of the class, have the defined structure. In Figure 1, a class at a high level is computer which has six attributes. The two subclasses of computer are desktop computers and mini & mainframe computers. Each class and its subclasses are individually a class. A rigid methodology does not exist for determining what classes should be modeled in a given situation. Though some heuristics exist, modeling is usually done by a designer with experience who has the foresight to plan an acceptable system.

Inheritance corresponds to the semantic modeling concepts of generalization and specialization (Smith & Smith, 1977). A subclass is a specialized case of the its superclass. We might observe in Figure 1 that the subclass *Desktop computer* is a specialization of the more general class *computer.* A subclass inherits themethods and attributes of all of its superclasses. This can potentially be a substantial time savings in software engineering as only the attributes and methods specific to a new subclass need be defined.

Not only are attributes predefined in a class, but so are the operations that are commonly referred to as methods, which give rise to the notion of encapsulation. Encapsulation is a desired attribute that occurs when the methods and the

Term	Description
Objects	Any real world entity which is associated with a system–wide unique identifier (Cattell, 1994; Kim, 1990).
Class	All objects which share the same set of attributes and methods may be grouped into a class. An object belongs to only one class as an instance of that class (Kim, 1990).
Attribute	An object has one or more attributes. The value of an attribute of an object is also an object (Cattell, 1994; Kim, 1990).
Methods	An object has one or more methods which operate on the values of the attributes (Cattell, 1994; Kim, 1990).
Persistence	Data remains after an application program or a user session is executed (Cattell, 1994).
Inheritance	Corresponds to the semantic modeling notion of generalization and specialization. Methods and class definitions can be inherited (Kim, 1990).
Encapsulation	Provides data independence through the implementation of methods, allowing the private portion of an object to be altered without affecting transactions that use the object type (Cattell, 1994).

Table 1: Object–Oriented Database Summary

structure of the object are "enveloped." That is, they are logically "surrounded." Encapsulation properly constructed means that objects encapsulate both the methods and structure and there is no means that either can be accessed except through the predefined methods.

A method is often defined such that it can be inherited by as many subclasses as possible; thus reducing the software development time. It is generally held that as many subclasses as possible should be able to inherit methods in order to reduce the software engineering effort as fewer methods will have to be constructed and tested.

Different methods can be invoked at many different locations even within the same subclass as long they do not conflict. An important issue for concurrency control is that if the definition of a method is being modified, no copy of it can be operating anywhere in the hierarchy. Therefore, an operation that updated a method would not be compatible with an operation that used that method to update an instance.

An example from Figure 1 is when a method is needed to modify an instance from the subclass **Apple Standard**. The attribute *Bundled Software* needs to be modified but the method that was created to modify *Bundled Software* is being updated. Modification of *Bundled Software* will have to wait until the method change is completed.

Table 1 contains a summary of the most crucial object–oriented database concepts.

Shortcomings of Current Concurrency Control Mechanisms for Object-Oriented Databases

In this section, a brief summary of the shortcomings of the O_2 and the ORION CC mechanisms are presented in order to demonstrate the need for improved CC mechanisms.

Concurrency control in the O_2 database system can be classified as a lock–based mechanism adapted to the object–oriented model. There are a number of similarities between the CC mechanisms of the O_2 and ORION database systems (Cart & Ferrie, 1992; Cattell, 1994). Not only do both systems use lock–based mechanisms, they both use implicit locking as a part of the object–oriented hierarchy in order to minimize the number of explicit locks that must be set (Gray & Reuter, 1993).

Both the O_2 and ORION CC mechanisms have limitations that can be improved upon. First, both have several different lock types including class definition reads and writes as well as instance read and writes. However, too many combinations in their respective compatibility tables evaluate to a **no**, or not compatible. By defining different lock types that are more specific along with additional lock types, more **yes** combinations are possible and a higher degree of concurrency is possible.

Second, both mechanisms do not distinguish class definitions from instances or method definitions. By treating different object types separately, the concurrency degree can be increased. Also, understanding the different operations that can occur with a different object type will enhance the concurrency degree.

Another approach to CC in implemented object–oriented databases is versioning (Cattell, 1994; Kim, 1990). Versioning allows many different versions of an object to be created and to exist without enforcing consistency requirements at creation time. Reconciling the value of an object to be consistent with public database must be done before it can be made public. Versioning is appropriate long–lived transactions but is not well–suited for standard, short–lived database transactions due to the amount of time the reconciling process takes.

To summarize, the current approaches to object–oriented database CC include a direct implementation of two–phase locking, versioning, and adaptations of lock–based algorithms that do not differentiate between object types. Versioning is acceptable for CC but is only used in specialized applications. A direct implementation of two–phase locking results in poor performance as object types are not treated differently. By developing a framework for object–oriented database CC, we highlight the areas to be addressed for an effective CC mechanism.

Framework for Object-Oriented Database Concurrency Control

In this section, a framework of object–oriented database concurrency control is developed which includes the necessary rules and dimensions that must be examined. As illustrated in Figure 1, **Computer** is the highest level class and is referenced as class 1. It has six attributes and three of the attributes are attribute classes themselves; namely, **Manufacturer**, **Secondary Storage**, and **Performance**. The other three attributes are considered simple attributes. The attribute classes are numbered 1.0.1, 1.0.2, and 1.0.3 to denote that they are attributes of class 1 and not subclasses. The heavy line is also used to signify the attribute class distinction.

In the narrative (outline form) that follows Figure 1, instances are listed for many classes. Though not explicitly stated, these instances would have a value for each attribute listed in the class. Thus, the Cray Supercomputer K instance would have data for each of the six attributes listed.

Two subclasses are defined for the class **Computer**; **Mini & Mainframe Computers** and **Desktop Computers**. They are numbered 1.1 and 1.2 respectively to designate that they are subclasses of class 1. Each of the subclasses has its own specific attributes but each subclass also inherits the attributes of the parent **Computer**. Thus, every instance of **Desktop Computers** would have 11 attributes that would possibly need to be completed; 5 of the attributes defined in its own class definition and 6 defined in the superclass, **Computer**.

Methods are similar to attributes in that each subclass has its own specific method defined specifically to operate on the attributes or instances within that subclass. Methods, like attributes can also be inherited. If an instance is being created at the **Desktop Computers** class, the attributes of the parent **Computer** (e.g. Manufacturer) will be inherited. The method for updating the attribute will have to be inherited along with the attribute. Thus, the new instance of **Desktop Computers** could be operated upon by possibly 11 methods; 5 of the methods defined in its own class definition and 6 defined in the **Computer** class.

Following this same logic then dictates that an instance of the class **Laptops** might have 23 attributes to complete which could potentially be operated on by 23 methods. Two items must be noted here though. First, some attributes in a general class might not apply to a subclass so the ability to not inherit attributes is often provided. For example, in many **Laptop computers** the ability to install a network card is not possible. For a given system then, programmers have the ability to disable the inheritance of the network card attribute in the laptop class.

The second important item is that it is possible to redefine a given attribute in a subclass from a superclass. This is accomplished by assignment of the same name to the attribute in the subclass as the attribute in the superclass. This is necessary because a superclass attribute might not quite capture the correct meaning for instances in a subclass. An example occurs in the **IBM Standard** class with the attribute {Performance}. An instance in the **IBM Standard** class does not inherit the {Performance} attribute from the class **Computer** but uses its own attribute. This is true because the computer reseller in this example wants to measure performance for IBM Standard personal computers by clock speed. Additionally, the **Laptops** class inherits the attribute {Performance} from the closest superclass (**IBM Standard**) in a conflict situation.

Dimensions Needed for Object–Oriented Concurrency Control

A CC mechanism in an object–oriented database system must address issues along four dimensions. They are as follows:

1.Hierarchical Level Dimension
2.Data Type Dimension
3.Composite or Complex Objects Dimension
4.Transaction Type Dimension

The intent of specifying these four dimensions is to provide the requisite framework for the necessary rules and lock types that follow in later sections. These dimensions are designed to be complete so that all issues of concurrency relate to at least one of the dimensions.

Hierarchical Dimension

The hierarchical dimension refers to the generalization/specialization property in object–oriented databases (Smith & Smith, 1977). As noted in Figure 1, the notion of a hierarchy is fundamental in object–oriented databases and concurrency control in a hierarchical model is discussed at length in Gray & Reuter (1993). A fundamental characteristic of concurrency control in a hierarchy is that a transaction updating an object is not at all independent due to the inheritance property. It might necessarily be true that updating an object at a high level in the hierarchy would require that objects at lower levels in the hierarchy also be locked. If not, inconsistencies could result. This is not an issue in the relational model as locking tables or records are independent. That is, if a table or a record needs to be updated, it is locked and there are no effects that occur elsewhere in the database.

An example from Figure 1 is a transaction updating the **Computer** class definition. A transaction that changes the attribute **Price (retail)** necessarily affects other classes in the hierarchy. In other words, a transaction updates the **Price**

(retail) attribute for instances by making a global change. The transaction might be increasing the **Price (retail)** attribute of all instances of computers in the entire database by 10%. This transaction cannot be allowed to execute concurrently with a transaction modifying the **Price (retail)** attribute for an instance in a specific subclass. One solution addressing the hierarchical dimension is discussed in Gray & Reuter (1993). Different lock types with specific definitions are presented in [9] and include intent, share, and exclusive.

An example of the hierarchical locking is when an intent mode lock is granted at a root node level before update mode locks are granted at the leaf node level. Then, if a conflicting transaction attempts to set locks on a parent node, it will be forced to wait until the initial transaction completes. If not, two conflicting transactions could set locks at different levels of the hierarchy and update the database with inconsistent data as shown in the example in the previous paragraph that included a retail price increase.

Data Type Dimension

Differences between the types of data that are being used in a transaction should be addressed in a CC scheme designed for object–oriented databases. Transactions that update instances, class definitions, aggregate data, and methods are fundamentally different.

In the Class Hierarchy Example in Figure 1, consider transaction T_1 that reads a value {Architecture_Type} (attribute A_3) from instance I_1 (COMPAQ Model A), of the class IBM Standard (class reference number 1.2.3). Also consider transaction T_2 that modifies the class definition of IBM Computers by adding an attribute {Video_Bus} (attribute 2). These two transactions are distinct as one is reading the value of an instance while the other is adding a class attribute. Under many CC schemes, these transactions would be incompatible as they are operating at the same class level. Though both of these transactions are operating at the same class level, they can be treated separately and thus, they are not incompatible. The results of exploiting the semantics of the data type results in higher levels of concurrency and higher levels of throughput.

Composite Dimension

A CC mechanism in an object–oriented database should facilitate composite objects. A composite object is defined as a collection of heterogenous objects to form a single object (Kim, 1990). The simple objects continue to be maintained in the system as separate entities, but the link to form a complex object significantly increases the flexibility of the database. Composite objects are commonly used in manufacturing applications: For example, cars (a composite object) are formed from many parts (which can be simple objects or in turn, complex objects formed from still more basic parts). The ability to form composite objects increases the flexibility for software engineers and thus, is significant in object–oriented databases.

If the definition of **Computer** in Figure 1 is changed in the system, it is necessary to not only lock the class definition of **Computer**, it is also necessary to lock the component objects of **Computer** to ensure consistency. The only component object of **Computer** is **Secondary Storage**. Kim, in [11] differentiates between component objects such **Secondary Storage** and a weak attribute such as **Computer_Manufacturer** (class 1.0.1). The distinction is useful as composite objects have the part–of relationship while weak attributes do not. Identifying the composite dimension as critical for concurrency control allows flexibility as well as an opportunity to increase performance. Proceeding to lock each individual object before the composite object can be locked would result in a degradation of performance. Thus, building a CC mechanism with composite objects specifically in mind will result in increased performance.

Transaction Type Dimension

Ultimately, an operation will perform one of two functions on an item of data; a *read* or a *write*. As explained in the section on two–phase locking, two separate operations can

Rule	Hierarchical Level Dimension	Data Type Dimension	Composite or Complex Objects Dimension	Transaction Type Dimension
#1	✓			✓
#2		✓		✓
#3		✓	✓	
#4		✓		
#5		✓		✓
#6	✓	✓	✓	✓
#7	✓	✓	✓	✓

Table 2: Rules to Dimensions Mapping

read the same item of data without resulting in inconsistencies. A potential problem exists when a write operation is performed on an item of data. If an item of data will be written by a given operation, no other operations can be allowed access to the data item (whether they be read or write operations) until the item of data is released by the initial write operation. If this were not true, we could observe lost update problems, dirty read problems, or phantom data problems (Bernstein, et al., 1987). It is thus important to distinguish between read and write operations in order to increase concurrency levels.

Rules of the Object–Oriented Concurrency Control Mechanism

Some basic rules are introduced in this section that O^2C^2 must follow in order to ensure consistency. These rules are designed to address issues related to the four dimensions previously discussed. As such, the rules as shown in Table 2 are listed along with the dimension to which it applies. It should be noted that some rules address issues related to more than one dimension. The purpose of these rules is to provide a basis for evaluating O^2C^2.

Rule 1 - Hierarchical, Transaction type. For instances that will be updated, the hierarchy must be locked in an upward direction with an intent lock type. This must occur so that no transaction could perform an operation at a very coarse level that would conflict with a transaction updating an instance (an operation at a fine level). An example from the class hierarchy is a transaction that updates the **Price** attribute for a **Desktop Computer**. This would conflict with a transaction acting at the **Computer** object level attempting to increase **Price** for all instances by 10%. Concurrency control must be operational so that these transactions do not produce inconsistent results. No locking will need to occur "down" the hierarchy though because of an instance update. The hierarchy will have an intent lock type that is either shareable or exclusive depending on whether the operation is a read or a write.

Rule 2 – Data Type, Transaction Type. Instances themselves can only be locked with an explicit read or write lock. Classes can be locked in a variety of manners including intent, shareable, and exclusive.

Rule 3 – Data Type, Composite. If the data type that is being updated is aggregate data, locks will need to be placed on all instances in the class and subclasses that form the aggregate data. There is no need to lock superclasses in the hierarchy as it is usually the case that aggregate data is stored or calculated at the highest level in the hierarchy where aggregation starts.

Rule 4 – Data Type. An operation on a class definition is considered to be separate from an operation on an instance. As new attributes are added, all the old instances must be updated to reflect the changes. Inconsistencies will not result because of updating a class definition simultaneously with creating instances using that class definition. Several instances may have to be updated if a class definition is altered because they will be out of date with the new structure, but the update will not make existing data inconsistent.

Rule 5 – Transaction Type, Data Type. Some transactions need to perform operations on all or most of the instances in a class. To minimize the number of locks that need to be held, a lock type that explicitly locks all instances of a class for one transaction is necessary. This lock type should exist for both read and write operations.

Rule 6 – Hierarchical, Data Type, Composite Objects, Transaction Type. Occasionally, a single transaction needs to perform operations on most of the instances in a class as well as class definitions. The same transaction may also need to calculate aggregate information or manipulate composite objects. It would be convenient to have one lock type that locks all the instances of a class, the class definition, and composite objects if any exist. This lock type should also handle either read or write operations.

Rule 7 – Hierarchical, Data Type, Composite, Transaction Type. An instance and a method are fundamentally different with respect to possible inconsistencies. An instance can be updated by one transaction simultaneously with the associated class definition update without causing inconsistencies. For example, when an attribute is added, old instances would not have values for the new attribute, but that does not cause data inconsistency. Inconsistencies could result however, if a method were updated at the same time it was being invoked by a transaction initiated at a specific class. This is true because a method is dynamic (it is being used by instances and is being read and updated by transactions affecting the class) and only one copy of the method is available. It is thus necessary to exclusively lock a method when it is in use by a class or an instance.

Lock Types of O^2C^2

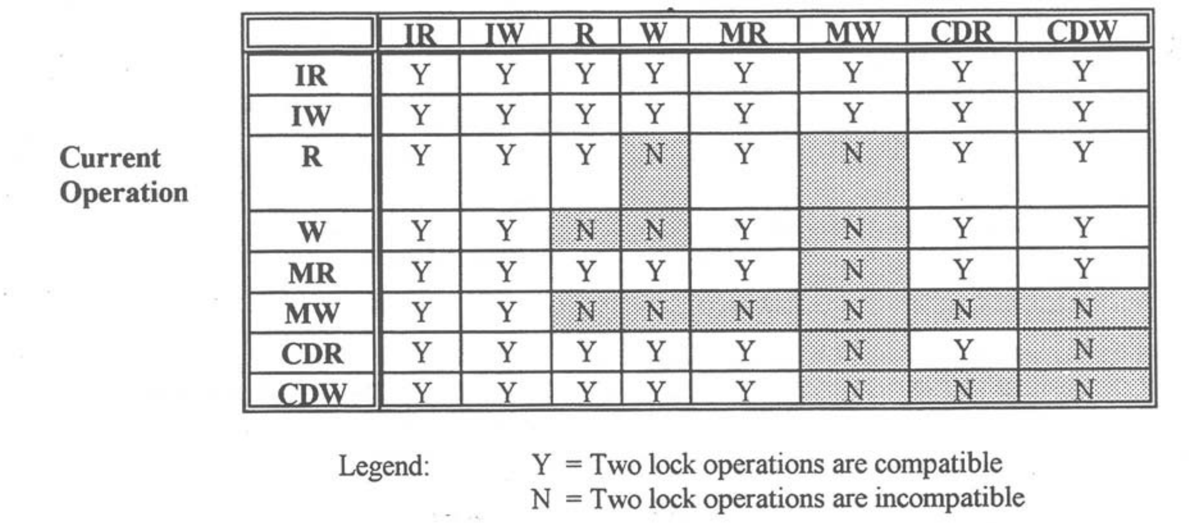
It is important to understand which operations are compatible with other operations in O^2C^2. After understanding the compatibility of various types of locks, it is important to understand how locks work in the hierarchy to effect concurrency control. Like other studies that examine hierarchical locking (Cart & Ferrie, 1992; Gray & Reuter, 1993; Kim, 1990), a compatibility matrix is presented in Table 4 after the lock types are explained in Table 3. As is the case in most compatibility matrixes, the current operation and the requested operation require the same item of data or the same class. For example, two transactions requesting an instance read (R) on the same instance are compatible whereas two operations requesting a write (W) operation on the same item data are not compatible.

Discussion

Some important points must be noted about the compat-

Lock Type	Description
Intention Read (IR)	This lock is placed on the class of an instance that is going to be read. All superclasses of the initial class are also IR locked.
Intention Write (IW)	This lock is placed on the class of an instance that is going to be updated. All superclasses of the initial class are also IW locked.
Read (R)	This lock can either be placed on a class or a single instance. If it is placed on a class, all instances of that class are read locked. This lock allows an instance to be read.
Write (W)	Whis lock can either be placed on a class or a single instance. If it is placed on a class, all instances of that class are write locked. This lock allows an instance to be updated.
Method Read (MR)	An intention read lock must be obtained on the method's class before this lock can be obtained on a method. This lock is needed when a method is going to be read.
Method Write (MW)	An intention write lock must be obtained on the method's class before this lock can be obtained on a method. This lock is needed when a method is going to be updated.
Class Definition Read (CDR)	An intention read lock must be obtained on the class before this lock can be obtained on the class definition. This lock is needed when a class definition is going to be read.
Class Definition Write (CDW)	An intention read lock must be obtained on the class before this lock can be obtained on the class definition. This lock is needed when a class definition is going to be read.

Table 3: O^2C^2 Lock Type Summary



Current Operation

	IR	IW	R	W	MR	MW	CDR	CDW
IR	Y	Y	Y	Y	Y	Y	Y	Y
IW	Y	Y	Y	Y	Y	Y	Y	Y
R	Y	Y	Y	N	Y	N	Y	Y
W	Y	Y	N	N	Y	N	Y	Y
MR	Y	Y	Y	Y	Y	N	Y	Y
MW	Y	Y	N	N	N	N	N	N
CDR	Y	Y	Y	Y	Y	N	Y	N
CDW	Y	Y	Y	Y	Y	N	N	N

Legend: Y = Two lock operations are compatible
N = Two lock operations are incompatible

Table 4: Compatibility Matrix of Transaction and Data Types for O^2C^2 Requested Operation

ibility matrix listed in Table 4 that shows the O^2C^2 lock types and their compatibility with one another. First, it has typically been standard that a write instance **(W)** lock type is not compatible with an intention to write lock **(IW)** from another transaction (Cart & Ferrie, 1992; Gray & Reuter, 1993; Kim, 1990). This is somewhat curious as it has also been standard that two **IW** locks on a class or a hierarchy of classes are compatible. In this proposal however, one transaction is able to hold a write lock on an instance while another transaction simultaneously holds an intention lock on the hierarchy where the updated instance exists.

Second, the difference between an instance and a method is crucial. This is highlighted in the compatibility chart where it is shown that an instance write **(W)** is compatible with a class definition write **(CDW)**. A method write **(MW)** however, is *not* compatible with a class definition write **(CDW)**. This is true for the reasons listed in rule number seven. The primary benefit of this addition is that concurrency will be increased as several instances can be accessed and updated simultaneously even though many transactions hold intent to write locks on the hierarchy. As was previously stated in the literature review, an increase in the number of transactions concurrently executing is a major goal in concurrency control research.

In the O^2C^2 mechanism, a class definition update is compatible with an instance update in the same class because of important assumptions concerning the system. The overriding concern governing the O^2C^2 mechanism is that the degree of concurrency be increased meaning that more "yes" answers in the compatible lock types matrix is a key objective.

An instance update operation can proceed concurrently with a class definition update in the same class. The instance update operation is given a copy of old the class definition that is publicly available. Once a class definition is updated, it becomes publicly available and all new instances use it. After all instance update operations that used an old class definition have either aborted or completed, the new class definitions are applied to all instances of that class.

The implications are that class definitions updates are allowed to proceed concurrently with instance updates. This is critical as class definitions are inherited and thus, if these operations were not compatible, concurrency would be decreased.

In this mechanism, a transaction holds an intent lock on a class hierarchy and then requests a write lock on the instance. This two level locking approach is necessary as it protects against a transaction setting a coarse lock on a hierarchy and then make conflicting changes to an instance that is concurrently being updated by a transaction operating at the instance level.

Achieving a high degree of concurrency in order to maximize performance is one of the two critical objectives of all CC mechanisms. The other is guaranteeing correctness. The discussion concerning the proof of correctness for the O^2C^2 mechanism is beyond the scope of this paper but the interested reader is directed to Olsen & Ram (1994).

Summary of Performance Analysis

The mechanism O^2C^2 has advantages over other object–oriented database CC mechanisms. First of all, it was designed with a comprehensive framework which included not only CC issues, but object–oriented database issues. In other words, the object–oriented database model was paramount in developing the framework. The second advantage is that O^2C^2 distinguishes between object types which increases the degree of concurrency. Transactions that would normally be blocked under other CC algorithms can proceed because of the object type distinction.

An excellent method of examining performance is through simulation studies. Empirical findings derived from simulation studies regarding the performance of O^2C^2 are reported in Olsen & Ram (1994). The simulation studies provides evidence that O^2C^2 is robust because it created correct schedules under a variety of conditions including conditions with a high number of transactions that caused thrashing to occur.

There are many interesting findings concerning the performance of O^2C^2. First, transaction throughput is much greater when a majority of the objects that are being read or updated are instance objects. As the number of method and class definition objects increases relative to the number of instance objects, transaction throughput is significantly reduced.

Another finding is actually a confirmation of other studies using the relational model that showed that as the number of active transactions increase, throughput increases to a point. Afterwards, thrashing effects are felt and throughput decreases. The fact that thrashing occurs comes as no surprise but it is interesting to note that regardless of the mix of object types, thrashing did not occur until 25% of the database was either read or write locked. This is a confirmation of what was observed in the relational model and demonstrates that O^2C^2 produces predictable results with regards to the shape of the throughput curve.

Third, increasing the number of transactions that update a given object decreases throughput. Indeed, as the percentage of write operations is increased, throughput decreases in an exponential manner at all levels of activity but was most pronounced at the highest levels of activity. Again, this is a confirmation of studies using other data models but it is important to note that predictable results occur.

Fourth, locking class definitions was the most detrimental to throughput as opposed to locking methods or instance objects. This is true because the inheritance notion dictates that a change to one class is inherited to the instances of all subclasses. Thus, locking a class definition can mean that many more classes and instances would also have to be locked. Knowing this, a database administrator might choose to schedule structural changes to the database during off–peak hours in order to maintain a higher level of concurrency and thus faster response times.

Conclusion

The O^2C^2 mechanism provides distinct advantages over other mechanisms and approaches in the object–oriented database area for certain systems. If the common transactions in a system are standard and short–lived, then the O^2C^2 mechanism provides higher levels of throughput, faster overall response times, and higher levels of concurrency. This translates into more satisfied users as they will not be waiting inordinate amounts of time for their transactions to complete. If the common transactions in a system are long–lived, then some type of versioning is appropriate.

References

Barghouti, N.S., & Kaiser, G.E. (1991), Concurrency Control in Advanced Database Applications, *Computing Surveys,* 1991, 269–317.

Bernstein, P.A., Hadzilacos, V., & Goodman, N.(1987), *Concurrency Control and Recovery in Database Systems,* Addison Wesley.

Cart, M., Ferrié, J., "Integrating Concurrency Control", in F. Bancilhon, C. Delobel, and P. Kanellakis, ed, Building an *Object–*

Oriented Database System: The Story of O_2, Morgan Kauffman Publishers, San Mateo, Ca., 1992.

Cattell, R.G.G.(1994), *Object Data Management*, Addison Wesley.

Eswaran, K.P., Gray, J.N., Lorie, R.A., & Traiger, I.L. (1976), The Notions of Consistency and Predicate Locks in a Database System, *Communications of the ACM*,19(11), 624–633.

Franaszek, P.A., Robinson, J.T., & Thomasian, A. (1992), Concurrency Control for High Contention Environments, *ACM Transactions on Database Systems*, 17(2), 304–345.

Garza, J.F., & Kim, W. (1988), Transaction Management in an Object–Oriented Database System, *Proceedings of the ACM SIGMOD International Conference*, 37–45.

Gray, J.N., Lorie, R.A., Putzolu, G.R., & Traiger, I.L.(1976), Granularity of Locks and Degrees of Consistency in a Shared Database, In *Modeling in Data Base Management Systems*. Amsterdam: Elsevier North–Holland.

Gray, J., & Reuter, A.(1993), *Transaction Processing: Concepts and Techniques*, Morgan Kauffman Publishers, San Mateo, Ca.

Hammer, M. & McLeod, D.(1981), Database description with SDM: A Semantic Data Model, *ACM Transactions on Database Systems*, 6(3).

Kim, W. (1990), Object–Oriented Databases: Definition and Research Directions, *IEEE Transactions on Knowledge and Data Engineering*, 2(3), 327–341.

Kung, H.T., & Robinson, J.T. (1981), On optimistic Methods for Concurrency Control, *ACM Transactions on Database Systems*, 213–226.

Olsen, D., & Ram, S. (1994), An Empirical Analysis of the Object–Oriented Database Concurrency Control Mechanism O^2C^2, Under Review, College of Business Administration, University of Akron, November 1994.

Papadimitriou, C.H. (1979), The Serializability of Concurrent Database Updates, *Journal of the Association for Computing Machinery*, 26(4), 631–653.

Smith, J., & Smith, D. (1977), Database Abstraction: Aggregation and Generalization, *ACM Transactions on Database Systems*, 2(2), 105–133.

David Olsen is an Assistant Professor of Accounting at the University of Akron. He received a B.S. degree in accounting at Fullerton State University in 1988, an M.S. degree in accounting from Brigham Young University in 1989 and a Ph.D. from the University of Arizona in 1993. Dr. Olsen's research includes database concurrency control in object-oriented, real-time active, and temporal databases. He is also researching the use of advanced database technologies to improve accounting functions. Additionally, Dr. Olsen is a member of ACM.

Sudha Ram is an Associate Professor of Management Information Systems at the University of Arizona. She has published articles in such journals as Communications of the ACM, IEEE Expert, IEEE Transactions on Knowledge and Data Engineering, Information Systems, Information Science, and Management Science. She has also presented her research at several conferences such as International Conference on Information Systems, International Conference on Data Engineering and other IEEE and ACM conferences. She was the guest editor for the December 1991 issue of IEEE Computer on Heterogeneous Distributed Database Systems. Dr. Ram's research deals with modeling and analysis of database and knowledge based systems for manufacturing, scientific and business applications. Her research on distributed databases has been funded by IBM, NCR, US Army, and NIST.