Dependency-based Concurrency Control in Logic-oriented Object Bases

L.D.H. Yang and Phillip C. Sheu
Department of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ 08855

ABSTRACT

A logic-oriented object base is a database that is constructed on an object data model. Using mathematical logic, a logic-oriented object base can be constructed to support classification, aggregation, generalization, and association. It further enhances the existing deductive databases with procedural semantics. In particular, it allows the preconditions and postconditions of object-oriented operations to be specified explicitly. Based on the preconditions/postconditions associated with the operations, the dependency information among different operations in a transaction can be derived. In this paper we describe a concurrency control mechanism that takes advantages of the dependency information among concurrent transactions. We show that higher level of interleaving can be obtained according to this mechanism.

1. Introduction

The idea of using First Order Logic to describe databases is not new. In deductive databases [GAMN84], logic has been used as the data definition language based on the relational data model. It is generally agreed that logical schemes can provide the following advantages [Brod84]:

(a) the availability of inference rules in terms of which one can define proof procedures,
(b) the availability of a clean, well understood, and well accepted formal semantics, and
(c) the availability of a natural way to assert queries.

However, an important drawback of logical schemes is the lack of organization principles for the facts that constitute a database [Brod84]. A second drawback is the difficulty in representing procedural knowledge. To cope with these problems, the framework of logic-oriented object bases has been proposed [Sheu86] to enhance the existing deductive databases with hierarchical organization principles and more procedural semantics. In particular, it facilitates the description of relevant entities in the application environment, collections of such entities, relationships (associations) among entities, inference rules, and queries to these entities. These kinds of knowledge are presented in a formal way and therefore can be used as a basis for intelligent reasoning.

The purpose of this paper is to consider the concurrency control problem in the context of logic-oriented object bases. In such systems, transaction semantics and integrity constraints can be stated explicitly. In particular, we consider the use of transaction semantics as well as integrity constraints such that concurrency control can be achieved in a more efficient way. Specifically, we show that if the precondition and postcondition of a transaction are known, it is feasible to interleave the operations of different (concurrent) transactions to a higher degree.

This paper is organized as follows. In Section 2 we first briefly review the essence of logic-oriented object bases. In Section 3 we describe the characteristics of the transactions in logic-oriented object bases. In Section 4 we introduce a scheduling criteria to interleave the operations of a set of transactions; the advantages provided by such mechanism are also illustrated.

2. Logic-oriented Object Bases

The impact of logic on relational databases is significant [GAMN84]. As an extension, we may expect a similar marriage between logic and semantic databases: logical representation can provide a formal representation for semantic data models and semantic data models can provide the organization principles for logical representation. Furthermore, coupled with the procedural semantic provided by PROLOG [Warr77], procedures can be included as integral parts of our conceptual schemes. Meanwhile, logic can also provide us a query language to query the semantic database. These concepts concepts constitute the core essence of the proposed logic-oriented object base framework.

Very briefly, a logic-oriented object base can be informally described as a database that is constructed based on an enhanced semantic data model that incorporates more procedural semantics and is represented in logic (see Figure 1). In a logic-oriented object base environment, a database is to be viewed as a collection of objects which are organized into classes. Each object or each class as a whole is described by a set of attributes. Associated with each object there is a set of methods defined; they are used to derive secondary attributes from the primary ones or they are used to update the object. Given an object base G, namely a set of classes and their associated methods, we define a logic-oriented object base be a tuple (A, G IC, DL), where A is a first order object language used to describe the objects, IC is a set of integrity constants, and DL is a set of deductive laws. The first order
language A consists of: a set of constraints (written in lower case letters) to name the objects, classes and attributes, a set of variables (written in upper case letters), and object-oriented knowledge representation. On one hand, the inheritance mechanism associated with the hierarchical knowledge structure provides a way to facilitate software reusability [Deut83]; on the other hand, integrity constraints in a logic programming system can provide the following advantages:

1. Arguments of methods can be typed.
2. Properties of methods can be described.
3. The consistency of the logic system can be maintained when changes are made.

3. Transactions in Logic-oriented Object Bases

In a logic-oriented object base, we define a transaction to be a formula of the form:

\[ \text{execute}(c : m(x_1, \ldots, x_n)) \]

where \( c \) is an object and \( m \) is a method predicate associated with \( c \). For each transaction, different from a transaction in a relational database, we may define its precondition that has to be satisfied before it can be executed and its postcondition that must be true after it has been executed. These are accomplished by introducing the following predicates:

1. \( \text{executable}(t) \), which is true if transaction \( t \) is executable.
2. \( \text{executed}(t) \), which is true if transaction \( t \) has been executed.

Generally, a transaction \( t \), its precondition \( p_1 \), and its postcondition \( p_2 \), can be asserted as an integrity constraint as follows:

1. \( \text{executed}(t) \rightarrow e_1 \)
2. \( p_1 \rightarrow \text{executable}(t) \)

Note that a transaction which is executable doesn’t mean it can be executed. When \( \text{executable}(t) \) is true it means transaction \( t \) is allowed to be executed if it is the only transaction in the system. When more than one transaction is executable, whether it can be executed or not should be determined by the concurrency control mechanism.

To illustrate the use of the above predicates, in a banking application we can define a transaction \( A: \text{withdraw}(M) \), where \( A \) is the name of the account and \( M \) is the amount of money withdrawn; the assert statements, the precondition and the effect associated with this transaction are illustrated below:

\[
\begin{align*}
\text{attribute_value}(A, balance, B) & \land (B \geq M) \rightarrow \text{executable}(A, \text{withdraw}(M)) \\
\text{if attribute_value}(A, balance, B) & \text{ then execute}(A, \text{withdraw}(M)) \leftarrow \text{delete}(\text{attribute_value}(A, balance, B)) \\
& \land \text{assert}([\text{attribute_value}(A, balance, B)])
\end{align*}
\]

In the following sections, we shall investigate how transaction interleaving can be facilitated with the provision of transaction preconditions/postconditions. Our future discussion will be based on the following assumptions:

(A1) When a transaction is defined, its associated precondition and postcondition are also defined with logical assertions. A transaction in the sim-

![Diagram](image-url)
plest case is a method, or it may be a set of methods that possibly reference different objects; in the latter case the component methods are called operations relative to the transaction. Consequently, in a composite transaction \( f = (f_1, ..., f_n) \) there may be separate sets of preconditions and postconditions associated with \( f \), \( f_1 \), ..., and \( f_n \).

(A2) For simplicity, we consider only those transactions whose components are methods. In other words, we don't consider nested transactions.

(A3) There are two types of conditions allowed to be associated with each method. The first kind of conditions are called inherent conditions, which are enforced by the system and are supposed to be followed in any transaction in which the method is involved. For instance, in the method \( puton \) in a block world has the following inherent pre-/post-conditions:

- If \( on(X,Y) \land on(X,U) \land on(Y,W) \) then \( execute(X, puton(Y)) \)
- \( execute(X, puton(Y)) \leftarrow delete(on(X,Y)) \land delete(on(X,U)) \land delete(on(Y,W)) \land assert(on(X,Y)) \land assert(on(X,U)) \land assert(on(Y,W)) \)

The other type of conditions are called user-defined conditions. They are the conditions a user wants to enforce in additional to the inherent conditions. For instance, in the method \( puton \) the operation \( s_{13} \) has a user-defined precondition \( attribute \_value(a, \text{amount}, C) \land (C \geq 80) \).

\[
\begin{align*}
&\text{execute}(f) \leftarrow \text{execute}(s_{11}) \land \text{execute}(s_{12}) \land \text{execute}(s_{13}) \\
&\text{attribute} \_value(a, \text{amount}, A) \Rightarrow \text{executable}(s_{11}) \\
&(\text{Operation } s_{11} \text{ is executable if the attribute } \text{amount} \text{ of } a \text{ has value } A) \\
&\text{execute}(s_{12}) \leftarrow \text{execute}(a, \text{add}(0,B)) \land \text{assert}(\text{attribute} \_value(a, \text{amount}, B)) \\
&(\text{To execute } s_{12}, \text{execute the method } a, \text{add}(0,B)) \\
&\text{execute}(s_{13}) \leftarrow \text{attribute} \_value(a, \text{amount}, C) \\
&(\text{After } s_{13} \text{ is executed, the attribute } \text{amount} \text{ of } a \text{ has value } C) \\
&\text{execute}(s_{12}) \leftarrow \text{assert}(\text{attribute} \_value(c, \text{amount}, 20)) \\
&(s_{12} \text{ is to assert } 20 \text{ as the value of the attribute } \text{amount} \text{ of } c) \\
&\text{execute}(s_{13}) \leftarrow \text{attribute} \_value(c, \text{amount}, 20) \\
&(\text{After } s_{13} \text{ is executed, } c \text{ has value } 20) \\
&\text{execute}(s_{12}) \land \text{attribute} \_value(a, \text{amount}, C) \land (C \geq 80) \Rightarrow \text{executable}(s_{13}) \\
&\text{execute}(s_{13}) \leftarrow \text{assert}(\text{attribute} \_value(a, \text{amount}, 50)) \\
&\text{assert}(\text{attribute} \_value(a, \text{amount}, 30)) \\
&\text{execute}(s_{13}) \leftarrow \text{attribute} \_value(a, \text{amount}, 30) \\
&\text{attribute} \_value(a, \text{amount}, C)
\end{align*}
\]

(A4) We assume that the items to be read in each operation are indicated in the precondition of the operation. For instance, for \( s_{11} \) in our earlier example, the precondition \( \text{attribute} \_value(a, \text{amount}, A) \) indicates that the attribute \text{amount} of the object \( a \) is to be written by \( s_{11} \). Note that the variable \( C \) here means an unknown value, although for some operations the values of the items to be written are known.

(A5) If the same variable \( C \) appears in the pre-/post-conditions of two different operations \( f_i \) and \( f_j \), the value of \( C \) is bound whenever it is instantiated. For instance, in the example described above, the value of \( C \) is bound after \( s_{13} \) is executed.

(A6) We assume that the operations in a transaction are not ordered. If a user-defined order needs to be enforced, it should be reflected in the preconditions associated with the operations. For example, if a user wants to have \( f_i \) executed before \( f_j \), one of the preconditions of \( f_i \) could be \( \text{executed}(f_j) \).

As another example, if \( f_i \) depends on the value of the attribute \( b \) of object \( a \), which is determined by \( f_j \), we can specify the following predicate as one of the postconditions of \( f_i \): \( \text{attribute} \_value(a,b,C) \). Also, we specify the following formula as part of the preconditions of \( f_i \): \( \text{executed}(f_j) \land \text{attribute} \_value(a,b,C) \) (e.g., operation \( s_{13} \) in example transaction \( f_1 \) above). Consequently, each transaction can be regarded as a set of production rules [DaKi76]. One of the major advantages provided by such production system representation is that operations in a transaction can be executed concurrently.

(A7) Each transaction or method \( f \) is represented as a triple \( (f, \pi_1, \pi_f) \), where \( \pi_1 \) is the precondition of \( f \) and \( \pi_f \) is the postcondition of \( f \).

4. The Transaction Scheduling Problem

In this section, we consider the use of transaction semantics as well as integrity constraints to facilitate concurrency control. Specifically, we show that if the precondition and the postcondition of a transaction are known, it is feasible that operations can be interleaved based on the dependencies among different operations such that concurrency can be achieved to a higher degree.

4.1. Concurrency Control in Conventional Databases

Concurrency Control (see, e.g., [KuRo81] [Bern79] [Papa79] [Schl78] [RoSL78] [KuPa79] [BeGo81] [Kohl81] [Eswa76] [Gray79]) is the activity of coordinating concurrent accesses to a database in a multiuser database management system. Research on this subject has been concentrating on synchronization techniques that will enable nonconflicting steps to be interleaved so that transactions can proceed with maximal concurrency. Most solutions to the access synchronization problem are based on some explicit or implicit locking scheme. In addition to the general-purpose two-phase locking scheme [Eswa76], there have been proposals for more flexible mechanisms that deal with specific applications such that higher level of concurrency can be obtained. These mechanisms either operate on data with specific structure (e.g., a tree) [Silb80]; only allow a set of simple operations on the data [Str81]; or may take as input semantic informa-
tion that specifies how transactions can be interleaved [Fish82] [Garc83] [Lync84].

4.2. A Dependency-based Scheduling Policy

Assuming that we are given a set of transactions, we are interested in interleaving their operations such that concurrency can be obtained.

Definition 1

Let \( T \) be a set of transactions \((f_1,p_1,e_1),..., (f_r,p_r,e_r)\) where transaction \( f_i \) has precondition \( p_i \), postcondition \( e_i \), and each \( f_i \) is composed of a set of operations \((f_{ij},p_{ij},e_{ij})\).... Let \( O_T \) be the collection of all the operations associated with the transactions, a schedule constraint \( \alpha \) of \( T \) is a partial ordering of \( O_T \).

Before a set of operations can be scheduled, we shall first derive a criterion to test whether a proposition is necessarily true in a situation. Of course a proposition is necessarily true in a situation if it is necessarily asserted. Once a proposition has been asserted, it remains to be true until denied. Thus a proposition \( p \) is necessarily true in a situation if it is necessarily asserted.

### Example

Assume that before \( f_1 \) and \( f_2 \) are executed, \( a \) is equal to 0. Now, if the amount of \( a \) exceeds or equals to 80, \( f_2 \) is executed.

### Definition 1

If we adopt the notation \( \rightarrow \rightarrow \) for partial implication, where \( p \rightarrow \rightarrow q \) is true if \( q = q_1 \wedge \cdots \wedge q_r \) and \( \exists i_1, \ldots, i_r, 1 \leq i_1 \leq r, p \rightarrow q_i, \) and \( p \rightarrow \rightarrow q \) is true if \( q = q_1 \wedge \cdots \wedge q_r \) and \( \exists i_1, \ldots, i_r, 1 \leq i_1 \leq r, p \rightarrow q_i \).

### Theorem

An operation \( f_{ik} \) cannot be fired if its effect negates (asserts) the precondition of another operation \( f_{jk} \) \((i \neq j)\) that has been enabled (disabled) by \( f_{ik} \), where \( f_{ik} \) has not been executed and \( f_{jk} \) is currently being executed or has been executed.

### Theorem

An operation \( f_{ik} \) cannot be fired if its effect negates (asserts) the precondition of another operation \( f_{jk} \) \((i \neq j)\) that has been enabled (disabled) by \( f_{ik} \), where \( f_{ik} \) has not been executed.

### Theorem

An operation \( f_{ik} \) cannot be fired if its effect negates the effect of another operation \( f_{ik} \) on which \( f_{ik} \) depends, where \( f_{ik} \) has been executed or is currently being executed.

### Theorem

An operation cannot be fired if it needs to read an item that is to be written by another operation which is currently being executed.

### Theorem

An operation cannot be fired if it needs to write an item that is to be written by another operation which is currently being executed.

Basically this policy states that there cannot be a dependency cycle in the resulting schedule. Consequently, any schedule that can be derived from this policy is deadlock-free [Coff71]. Note that the effects associated with each operation may not be known until the operation is executed (e.g., in our earlier example, the value \( C \) of the effect \( \text{attribute value}(a, \text{amount}, C) \) associated with \( s_{11} \) cannot be determined until \( s_{11} \) has been executed). In this situation, some operations may be allowed to be executed but may need to be undone if later it is determined not fireable based on its effects.

### Example

The above algorithm can be illustrated by a simple example. Assume that there are two transactions \( f_1 \) and \( f_2 \). Transaction \( f_1 \) adds 10 to the attribute \( \text{amount} \) of object \( a \), assigns 20 to the attribute \( \text{amount} \) of object \( c \), and if the amount of \( a \) exceeds or equals to 80, adjusts the attribute \( \text{amount} \) of \( a \) to 50 and adjust the attribute \( \text{amount} \) of \( c \) to 30. Transaction \( f_2 \) assigns 50 to the attribute \( \text{amount} \) of \( a \).

Note that the operation \( s_{13} \) of \( f_1 \) is executable as long as the amount of \( a \) is greater than or equal to 80. Now, assume that before \( f_2 \) and \( f_2 \) are executed, \( \text{attribute value}(A, \text{amount}, c) \) is true. According to the scheduling policy described above, the following
schedule: \[ s_1 \rightarrow s_2 \rightarrow s_3 \]
is forbidden by the scheduler since the effect of \( s_2 \) will negate the precondition of \( s_3 \), which has been previously enabled by \( s_1 \). In other words, no interleaving is possible in this example. However, if we change \( f_2 \) to:

\[
\begin{align*}
\text{execute } (f_2) & \rightarrow \text{execute } (t_3) \\
& \rightarrow \text{assert } (\text{attribute } (x, \text{amount}, 90)) \\
& \rightarrow \text{assert } (\text{balance } (x, \text{amount}, 90)) \\
& \rightarrow \text{execute } (t_5) \\
\end{align*}
\]

then the above schedule is permitted by the scheduler, although it is forbidden by any other lock-based scheduling scheme [Bem86].

4.3 Serializability and Semantic Transaction Types

Although the requirement of serializability has been widely accepted as an important correctness criterion for databases, there have been arguments criticizing it to be a requirement that is too strong for many applications (see, e.g., Lync84). We shall use a simple example [MoCo84] to illustrate this.

Example

Suppose that there are three bank accounts: a, b, and c. Accounts a and b will be customer accounts that are charged a 10 dollar penalty fee if they ever go below 1500 during a month. c will serve for the bank's internal accounting and records the total penalties collected. The only restriction on accounts will be the money be accounted for, i.e.,

\[
\text{Bal}(a) + \text{Bal}(b) + \text{Bal}(c) = \text{tot},
\]

where \( \text{tot} \) is a variable that records the total amount of money in the bank, and \( \text{Bal}(X) \) is the balance of account \( X \). The remaining two variables \( \text{pena} \_\text{coll}(a) \) and \( \text{pena} \_\text{coll}(b) \) are flags which become false if balance is below 1500 and ensure that the penalty fee is collected just once a month. It will, in fact, be collected the first time the account goes below 1500. A special transaction, of type \( \text{rst} \) (see below), which resets both flags to their initial value ('false'), will be run at the bank's closing time on the last day of the month. Note that this has to be the last transaction, with access to the flags, submitted to the database during the course of any month.

For this database we will define the following three types of transactions:

(a) \( d2(X) \) (deposit in both accounts a and b the amount \( X \)). Its steps are:

i) \( \text{Bal}(a) \Rightarrow \text{Bal}(a) + X; \text{tot} \Rightarrow \text{tot} + X; \)

ii) \( \text{Bal}(b) \Rightarrow \text{Bal}(b) + X; \text{tot} \Rightarrow \text{tot} + X \)

(b) \( w2(Y) \) (withdraw from both accounts a and b the amount \( Y \)). Its steps are:

i) \( \text{Bal}(a) \Rightarrow \text{Bal}(a) - Y; \text{tot} \Rightarrow \text{tot} - Y; \) if \( \text{Bal}(a) < 1500 \) and \( (\text{pena} \_\text{coll}(a) = \text{false}) \) then

\( \text{Bal}(a) \Rightarrow \text{Bal}(a) - 10; \text{Bal}(c) \Rightarrow \text{Bal}(c) + 10; \text{pena} \_\text{coll}(a) \Rightarrow \text{true} \)

end if.

ii) \( \text{Bal}(b) \Rightarrow \text{Bal}(b) - Y; \text{tot} \Rightarrow \text{tot} - Y; \) if \( \text{Bal}(b) < 1500 \) and \( (\text{pena} \_\text{coll}(b) = \text{false}) \) then

\( \text{Bal}(b) \Rightarrow \text{Bal}(b) - 10; \text{Bal}(c) \Rightarrow \text{Bal}(c) + 10; \text{pena} \_\text{coll}(b) \Rightarrow \text{true} \)

end if.

(c) \( \text{rst} \) (reset both flags):

\( \text{pena} \_\text{coll}(a) \Rightarrow \text{false}; \text{pena} \_\text{coll}(b) \Rightarrow \text{false} \).

Suppose now that we initially have \( \text{Bal}(a) = \text{Bal}(b) = 2000 \) and \( \text{Bal}(c) = 0 \), and that \( T_1 \) and \( T_2 \) are of types \( \text{d2}(500) \) and \( \text{w2}(800) \) respectively. Consider the following execution schedule \( s \):

\( T_1 \) step (i);

\( T_2 \) step (i);

\( T_2 \) step (ii);

\( T_1 \) step (ii).

Schedule \( s \) is a valid interleaving of the steps. To check this we can see that at the end of the execution \( \text{Bal}(a) = 1700; \text{Bal}(b) = 1690; \text{Bal}(c) = 10 \) and \( \text{tot} = 3400 \), which fulfills the restriction that \( \text{Bal}(a) + \text{Bal}(b) + \text{Bal}(c) = 3400 = \text{tot} \). Observe that any serial schedule of these two transactions would either have produced \( \text{Bal}(a) = \text{Bal}(b) = 1700 \) or \( \text{Bal}(a) = \text{Bal}(b) = 1690 \). From this simple example, we can see that a non-serializable but consistent schedule is possible. Note that not having to enforce serializability could be an advantage. In the schedule above, \( T_2 \) can access the funds it needs without waiting for \( T_1 \) to finish.

As suggested in [Lync84], steps of different application database transactions might be allowed to interleave in various ways; the set of allowable interleavings is determined by the application represented. At one extreme, it might be required that all allowable interleavings be serializable. At the other extreme, the interleavings might be unconstrained. To facilitate the above arguments, transactions are divided into steps in [Garc83] [Lync84]; each step is a collection of conventional database operations that will be performed as an atomic unit. Two transaction's are compatible if their steps can be interleaved arbitrarily without violating database consistency. The users of the database define what transactions are compatible by classifying them into semantic types; and for each type, the compatible transaction types are given. Specifically, the set of transactions that are compatible with a transaction \( f \) can be described by its compatibility set \( cs(f) \). For example, the declaration
means transactions of type x can be interleaved with transactions of type y and with those of type z. However, transactions of types y and z cannot be interleaved with each other. In the above bank example, since the addition operation is commutative, and account c will be credited 10 dollars every time the penalty is charged, then it is clear that interleaving of the steps of transactions of type d2 and w2 will not violate the restriction Bal (a) + BAL (b) + Bal (c) = tot. Consequently, we can define cs (w2) = { (d2, w2) }. By the same facts and since transactions of type d2 do not access the variables pens (a) and pens (b), then the compatibility set of d2 can be defined as cs (d2) = { (d2, w2), (d2, rst) }. Running transactions of type w2 and rst concurrently could result in collecting a penalty fee twice and providing the incorrect flag value for the beginning of the next month. For this reason we define cs (rst) = { (d2, rst) }.

We think that the dependency-driven scheduling mechanism discussed above provides an appropriate means to achieve this flexibility. We shall illustrate this with the same example introduced above. As the first step, we convert the three types of transactions d2, w2 and rst into the following form:

Consequently, the dependency-based scheduling mechanism generates the same behavior as described earlier without specifying the compatibility sets. Furthermore, higher degree of interleaving can be achieved if we slightly modify the preconditions of s22 and s24 as shown below. This is because the constraints (i) and (ii) in observation (O2) may be relaxed (i.e., a small deposit may not be able to save the penalty if a large withdrawal precedes it) if we remove the data dependencies between s21, s23 and between s21, s24. For example, suppose now that we initially have Bal (a) = Bal (b) = 2000 and Bal (c) = 0, and t1 and t2 are of type d2 (2000) and w2 (6000) respectively. The following execution schedule s:

The following observations can be inferred from the scheduling policy:

1. Given two transactions t1 = (a, d2) and t2 = (b, d2), the operations in t1 and t2 can be interleaved arbitrarily.
2. Given two transactions t1 = (a, d2) and t2 = (b, d2) then, the execution in t1 and t2 can be interleaved arbitrarily except that:
   (i) operation t1 - r1 cannot be executed between operations t1 - s1 and t2 - s2 since the execution of t1 - r1 will change the value of Bal.
   (ii) operation t1 - s1 cannot be executed between operations t1 - s2 and t2 - s2 since the execution of t1 - s1 may change the value of Bal.
3. Given two transactions t1 = (a, d2) and t2 = (b, w2) then, the execution in t1 and t2 can be interleaved arbitrarily except that:
   (i) operation t2 - s1 cannot be executed between operations t1 - s1 and t2 - s2 since it may change the value of pens (a).
   (ii) operation t2 - s1 cannot be executed between operations t1 - s1 and t2 - s2 since it may change the value of pens (b).
4. Given two transactions t1 = (a, d2) and t2 = (b, w2) then, the execution in t1 and t2 can be interleaved arbitrarily except that:
   (i) operation t1 - s1 cannot be executed between operations t1 - s1 and t2 - s2 since it may change the value of pens (a).
   (ii) operation t2 - s1 cannot be executed between operations t1 - s1 and t2 - s2 since it may change the value of pens (b).
5. The predicate execute(tot: subtract (2Y)) is true if 2Y is subtracted from the attribute value of the object tot.
4.4 Comparison with Two-Phase Locking Schemes

As we have mentioned earlier, most solutions to the concurrency control problem in databases are based on some sort of locking schemes [Bern86]. A transaction or atomic action may lock objects to ensure their accessibility while in a temporarily inconsistent state. A detailed description of the existing locking schemes can be found in [Bern86]. In the following we shall compare the dependency-driven scheduling scheme with the lock-based concurrency control scheme that is based on the simplest lock-unlock transaction model.

In the lock-unlock transaction model, a transaction is viewed as a sequence of lock and unlock statements. Each item locked must be subsequently be unlocked. Between a step lock A and the next unlock A, a transaction is said to hold a lock on A. Apparently, the precondition for each lock A statement is that A is not locked. Consequently, a lock A statement depends on the most recent unlock A statement. According to the scheduling policy, the following behavior can be observed:

(a) No lock A statement can be executed in a transaction between a pair of unlock A and lock A statements in another transaction.

(b) When an item is locked, no other transaction can lock it again.

Note that when all the items involved in a transaction are locked more than once, and at least one of these locks happens after all items have experienced a lock, the schedule becomes 2-phase. This can be illustrated by the simple transactions shown in Figure 3. The lock on A (B) is held by T until the last unlock A (B) statement. Since the last lock A (B) statement happens after all the items have experienced a lock, it is equivalent to the two-phase locking scheme, where a lock cannot be released until all the items involved in a transaction have been locked. However, if only few items are locked more than once, then concurrency can be achieved to a higher degree. For example, as shown in Figure 4, let transaction t1 = <(lock A) (unlock A) (lock B) (unlock B) (lock C) (unlock C)> and transaction t2 = <(lock B) (unlock B) (lock D) (unlock D)>; then the following schedule is not two-phase (since lock B in t1 is granted before lock C in t1 is granted) but is legal according to the dependency-based scheduling policy:

4.5 Implementation of the Scheduling Policy

To implement the above scheduling policy, there are two obvious solutions:

(a) The scheduling is done by a centralized scheduler, or

(b) Every transaction keeps track of the status of other transactions. Also, every transaction should be informed of the state of the system continuously.

Unfortunately, the first approach will create a performance bottleneck, and the second approach will involve too much overhead. To avoid these problems, we propose the following approach:

(1) In addition to the integrity constraints that have to be held all the time when the system is operating, we create a temporary integrity constraint list. In this list, temporary constraints to interleave con-
current transactions are created and removed dynamically. When a constraint is included in the list, no active transaction is allowed to violate the constraint. If a step within a transaction is blocked by an active temporary constraint, it remains to be blocked until that constraint is removed from the list.

2. If \( f_{ik} \) depends on \( f_{iu} \) in transaction \( f_i \), then the condition \( f_{ik} \) relies on (which is enabled by \( f_{iu} \)) will be included in the temporary integrity constraint list.

3. If the effect of certain action \( f_{jc} \) violates one of the constraints in the temporary integrity constraint list, then \( f_{jc} \) will be blocked until the blocking constraint is removed.

4. If \( f_{ik} \) depends on \( f_{iu} \) in transaction \( f_i \), then the condition \( f_{ik} \) relies on (which is enabled by \( f_{iu} \)) will be removed from the temporary integrity constraint list after \( f_{ik} \) is executed.

5. Conclusion

Although the subject of knowledge base has attracted extensive attention recently, the problem with concurrency control and fault tolerance in knowledge base systems has seldom been attacked. In this paper we have described a policy to schedule concurrent transactions in which the preconditions and effects associated with the transactions can be specified. We have also proposed a recovery procedure when transactions failures are detected.

The advantages of the scheduling algorithm can be summarized as follows:

(1) For conventional transaction models, it is less constrained than the two-phase protocol, particularly for short transactions.

(2) It can perform concurrency control based on semantic dependencies.

It can be expected that many other issues in conventional databases need to be reconsidered in knowledge bases. One typical example is the fault tolerance problem in distributed environments. We believe that this system issues are critical to the success of next-generation databases.

6. References