Concurrency Control in Object-Oriented Databases

Wai Lam, Yalin Wang, and Yongbing Feng

Contents

1 Introduction

Transaction processing is concerned with controlling the way in which programs share a common database. Normally, a *transaction* is viewed as both the unit of *concurrency* and the unit of *recovery* for database applications. As a unit of *concurrency*, the steps of several transactions can be interleaved so that they will not interfere with each other. As a unit of *recovery*, a transaction either succeeds totally or it has no effect on the database. The system always recovers to the boundaries of a transaction, such that results of partially completed transactions will not be visible.

In traditional transaction management, a transaction is a sequence of reads and writes against a database. A transaction as used in conventional applications has two properties: *atomicity* and *serializability*. The atomicity property means that the sequence of reads and writes in a transaction is regarded as a single atomic action against the database. It ensures that if a transaction cannot complete, the system backs out all writes which may have been recorded in the database. Or, if a transaction completes successfully, the system guarantees that all writes are recorded in the database. The serializability property means that the effect of the concurrent execution of more than one transaction is the same as that of executing the same set of transactions one at a time serially. It ensures that a transaction is completely shielded from the effects of any other concurrently executing transactions.

The atomicity and serializability properties of transactions have been highly useful for conventional business data processing applications. However, database systems have improved the application-development process in large data-intensive environments by providing a single, uniform view of data and the availability of high-performance graphics workstations has increased the breadth and complexity of data-intensive applications that are being attempted. Examples of such environments, are computer-aided design (CAD), computer-aided software engineering (CASE), and office information system (OIS). All of these are characterized by long and collaborative transactions in which concurrency must be optimized for adequate overall performance. However, the *atomicity* and *serializability* of conventional transactions have highly undesirable consequences for long-duration transactions (that is, transactions whose duration is much longer than that of conventional transaction). The *atomicity* property means that if a long-duration transaction cannot complete, all the work that has been done by the transaction must be backed out. The *serializability* property means that if a long-duration transaction holds a lock on a object, any other long-duration transactions that must access the same object in a conflicting mode must be blocked until the first long-duration transaction completes.

Concurrency is optimized using application semantics and data semantics. Hence, semantics approaches can be broadly divided into two groups, namely the *transaction approach* and the *data approach*. Models of the transaction approach define concurrency properties on transactions according to the semantics of the transaction and the data they manipulate. Models of the data approach define concurrency properties on abstract data types according to the semantics of the type and its operations. In Section 2, we describe three models of the transaction approach. The first one is the compatibility set approach [GM83] which is one of the early attempts to deal with concurrency control through transaction semantics. Another one is the constraint-based approach [KB88] which can handle nested transaction environment common in object-oriented databases. The third model is due to a recent work [SZ89] [NZ90] which makes use of *patterns* to express correctness constraints allowing higher concurrency control. In Section 3, we describe two models of the data approach. The first model [Sch84] redefines a transaction as a sequence of typed operations on objects. The second model [HW88] uses a serial dependency relation as the basis for defining operation conflicts.

2 Transaction Approach

2.1 A Design Task Example

We carefully make use of a design task environment example to illustrate different models using transaction approach for handling concurrency transactions in object-oriented databases. Consider a computer-aided software engineering (CASE) tool being used in developing a sophisticated manuscript system. An object-oriented database is used to support interactive design transactions in the CASE environment. One subsystem of the manuscript system is a word processing system which has three modules. The *Display* module manages the user interface, the *Processing* module maintains the document in its internal format according to the commands issued from the *Display* module, and the *Storage* module deals with disk I/O. Each module has a designer responsible for designing and programming it by issuing a long-duration design transaction from a workstation. These three modules for the word processing subsystem form a group under the whole development project.

Associated with the word processing subsystem, two classes of objects are shown as follows:

```
class     | IF-SPEC                PROGRAM-SPEC
          |
attribute | specification-name     program-name
          | calling-routines       input-arguments
          | called-by-routines     output-arguments
          | external-var                  :
          |      :                        :
          |      :
```

The objects in the classes *IF-SPEC* and *PROGRAM-SPEC* are the interface and program specification respectively and may be modified by different modules. Some transactions that will be run on this word processing subsystem are of the following categories:

```
INIT        : initialize the database
REPORT      : read the detail of each object and print a report
DISPLAY-DVP : design and develop the display module interactively
PROCESS-DVP : design and develop the processing module interactively
STORAGE-DVP : design and develop the storage module interactively
```

2.2 Compatibility Set Approach

One of the early attempts to deal with concurrency in databases by the transaction approach can be found in the work of Garcia-Molina [GM83]. It utilizes semantic knowledge of an application to process transactions efficiently in distributed database environment. One important characteristic of transactions in the distributed system is that they are of long duration due to relatively large communication delays in the network. Hence, a different mechanism is proposed to handle these long transactions which can be applied to transactions in object-oriented database settings. The main idea is to allow nonserializable schedules which preserve consistency and are acceptable to the users.

The approach starts off by categorizing the transactions into different types according to their semantic information. Furthermore, each transaction is divided into atomic steps so that these steps can be interleaved without violating the consistency of the database. We illustrate this idea through a simple example. Consider the word processing system described above. Suppose there are two more attributes for the class *PROGRAM-SPEC*. One is *estimated-cost* which represents the estimated cost needed for the development. The other is *complexity* which has the possible values, "high" and "low". Initially, the cost starts with 0 and the complexity starts with "low". If the estimated cost exceeds a certain limit, the complexity flag will be changed to "high". It is left as "high" even if the cost, due to some adjustment in the future, drops below the limit. Now consider a transaction which increments the cost of two related programs MO1 and MO2. Let IC (Increment Cost) represent all transactions of this type. Each transaction of type IC performs two atomic steps.

```
IC :
    step 1 - increment MO1.estimated-cost by 1.
             if MO1.estimated-cost > 10 then let MO1.complexity = "high"

    step 2 - increment MO2.estimated-cost by 1.
             if MO2.estimated-cost > 10 then let MO2.complexity = "high"
```

Another type of transaction DC (Decrement Cost) is to decrement the estimated cost of MO1 and MO2 due to adjustment. It also consists of two atomic steps as follows:

```
DC :
    step 1 - decrement MO1.estimated-cost by 1.
    step 2 - decrement MO2.estimated-cost by 1.
```

If the steps of transactions IC and DC are interleaved arbitrarily, it can be shown that some schedules are non-serializable. For example, let the estimated cost of both objects be 10 and the complexity be "low". A schedule like:

IC.step1 → DC.step1 → DC.step2 → IC.step2

would leave the database in the state as follows:

```
MO1.estimated-cost = MO2.estimated-cost = 10
MO1.complexity = "high"
MO2.complexity = "low"
```

This state could have never been produced by a serial execution of two transactions, but it is consistent with the semantic of the objects.

More importantly, by relaxing the serializability restriction, it allows a higher degree of concurrency while still allowing the result to be semantically consistent with the application. Based on this idea, a *compatibility set* is associated with each transaction type. Each compatibility set is composed of *interleaving descriptors* which depict a specific type of allowable interleaving. For instance, consider the four transaction types of the word processing system given in the previous subsection. Assume that the steps in transactions of type DISPLAY-DVP and PROCESS-DVP can be interleaved arbitrarily without violating the consistency. The compatibility set cs(DISPLAY-DVP) = cs(PROCESS-DVP) = {DISPLAY-DVP, PROCESS-DVP}. But the steps of the transactions of INIT and REPORT cannot be mixed with any other types of transactions except themselves. Therefore, cs(INIT) = {INIT} and cs(REPORT) = {REPORT}.

Locking is used for the implementation of this approach. It utilizes two kinds of locks, namely *local locks* and *global locks*.

Local locks are used to guarantee that the transaction steps are executed as atomic actions. Global locks are used to ensure that the interleavings of atomic steps do not violate consistency. Associated with each global lock on object O will be an *interleaving descriptor* ID(O). This descriptor indicates that one or more transactions of type ID(O) have accessed the locked object, are being interleaved, and have not yet finished.

When a transaction needs to access an object O, it must first obtain a global lock for O and then a local lock for O.

2.3 Constraint-Based Approach

Interactive and cooperating design applications such as CAD, CASE, and OIS are the typical applications of object-oriented database systems. A CAD environment requires a significantly different model of transaction from that developed for typical data-processing applications. Constraint-based model [BK85] [KB88] is a general model of transactions and can be specialized to a CAD environment.

2.3.1 Intuitive Model of CAD Transaction

We lump together design, engineering, and software development projects under the term *design projects*. Now, let's consider the *design projects*.

First, a large design effort is typically partitioned into a number of mostly non-overlapping projects which consist of design data and some data about design data. Second, through windowing facilities, a designer may create and manipulate multiple windows, executing multiple tasks concurrently. Third, each project has a number of designers who further subdivide the project into a number of subtasks. Fourth, in a complex design project, it is often the case that some tasks are subcontracted to other (group of) designers.

We can combine the four observations to derive the intuitive model of CAD transactions. A design environment consists of a number of *project transactions*, each of which may consist of a *set* of *cooperating transactions*. Each *cooperating transaction* is a *hierarchy* of *client/subcontractor transactions*. Each *client/subcontractor transaction* is a *set* of *designer's transactions*, which in turn is a *set* of *short-duration transactions*. A *short-duration transaction* is initiated from a window of the designer's workstation.

2.3.2 Formal Definitions of CAD Transaction

A *global database* consists of a public database and some private databases of active transactions. We refer to a particular set of values taken on by the global database as a *state* of the database. The set of all possible states is denoted by S.

definition: An *integrity constraint* C is a predicate on S

$$C : S \ \rightarrow \ \{ \text{ TRUE, FALSE } \}$$

The subset of S consisting of states that satisfy constraint C is denoted by S_c.

$$S_c = \{s \mid s \in S \,\text{and}\, C(s)\}.$$

Each transaction in the model is required to preserve some integrity constraint.

definition: A *database operation* (or *operation*, for short) O is a mapping from S to S:

$$O : S \rightarrow S.$$

In the following, we use M to denote the set of all operation names.

Examples of operations are: (1) read or write of a private database, (2) read or write of the public database, (3) read, write or update of indices for a database (public or private).

definition: A *transaction* is a 3-tuple (T, O, C) where

- T is a set of transactions or operations
- O is a partial order on T
- C is the integrity constraint preversed by the transaction

We require that for all transactions (t, o, c) in T, C implies c. In other words, nested transactions may have weaker constraints than their ancestors, but not vice-versa.

definition: The *closure* $(T, O, C)^\star$ of a transaction (T, O, C) is the transaction $(T^\star, O^\star, C)$ where

- $T^\star = (M \cap T) \cup (\bigcup_X (t', o', c)^\star)$ and X = $\{ \ (t', o', c) \mid (t', o', c) \in T - M \ \}$
- $O^\star$ is the partial order induced on $T^\star$ defined as follows: Let a and b be operations in $T^\star$. In $O^\star$, $a < b$ if one of the following holds:
    1. $a, b \in T$ and $a < b$ in O

2. There exist a pair of transactions (t_1, o_1, c_1) and (t_2, o_2, c_2) such that $a \in t_1{}^\star, b \in t_2{}^\star$ and $t_1 < t_2$ in O
3. There exists a transaction (t_1, o_1, c_1) such that $a, b \in t_1{}^\star$ and $a < b$ in $(t_1, o_1, c_1)^\star$

The closure of a transaction is a "flattening" of the transaction hierarchy.

definition: An *execution* e of a transaction $t = (T, O, C)$ is a transaction $(T^\star, O', C)$ where O' is a total order such that O' implies $O^\star$. That is, if $a, b \in T^\star$, and $a < b$ in $O^\star$, then $a < b$ in O'. We denote the set of all executions of (T, O, C) by E_t.

An execution of a transaction represents a linear (total) ordering of all operations of a transaction that is compatible with the partial orders defined at the various levels of nesting in the transaction. The concept of execution is similar with that of *schedule* in conventional transactions. The effect of an execution e is a mapping from S to S defined as follows:

$$e : m_1 \circ m_2 \circ \ldots \circ m_n$$

where $m_1, m_2, \ldots, m_n$ be the ordered list of operations in an execution e.

definition: An execution e of (T, O, C) is *correct* if for all states $s \in S_C, e(s) \in S_C$.

It is the responsibility of the system to ensure that, given an integrity constraint C, only correct executions are permitted. A protocol is a set rules that restricts the set of admissible executions.

definition: A *protocol* P is a mapping from the set E_t of executions of t to {true, false}. If e is an execution, then $P(e)$ if and only if e is an execution *legal* under P. A protocol P is *correct* if for all executions $e \in E_t$, $P(e)$ implies e is correct. We say that a protocol is *nested* if it can be expressed in terms of the partial orders of transactions contained within a nested transaction.

We do not require a protocol to capture all possible correct executions since such a requirement is not feasible in practical systems.

2.3.3 Formal Descriptions of Two Kinds of Transactions

Now, as examples of the formal definitions of the model, we give the descriptions of two kinds of transactions – cooperating transactions and client/subcontractor transactions.

In the simplest case in which cooperating transactions consist solely of database operations, a complete definition of a set (T, O, C) of cooperating transactions is as follows:

- T is a set of transactions of the form (t, o, c) where:
  - t is a set of transactions of the form (t', o', c') where:
    - t' is a set of operations
    - o' is some total order

    * c' is TRUE
  - o is some partial order
  - c is TRUE

- O is the empty order
- C is the set of consistency constraints that we wish to enforce on the entire database.

In general, each transaction within a cooperating transaction (that is, a designer's transaction) is a hierarchy of clients/subcontractors. This situation is represented by (T_c, o, c) where:

- T_c is a set of operations and subcontractors
- o specifies when each subcontractor is initiated relative to the other members of T_c
- c is some local consistency constraint.

Let (T_c, O, C) be a transaction where:

- T_c contains T_s, where T_s represents a subcontractor of (T_c, O, C) (that is, (T_c, O, C) is a client of T_s). $T_s = (t', o', c')$
- O is some partial order
- C is a set of integrity constraints that we wish to enforce on the entire database.

We say that T_s is a *subcontractor* of *client* T_c under protocol P if $P(e)$ is true only if e is equivalent to a correct execution of $((T_c - T_s) \cup t', O'', C)$, where O'' contains:

- All elements $a < b$ of O such that $a, b \in T_c - T_s$
- All elements $a < b$ of o'
- All orderings of the form $a < b$, where $a \in T_c - T_s$ and $a < T_s$ appears in O, and $b \in t'$
- All orderings of the form $a < b$, where $b \in T_c - T_s$ and $T_s < b$ appears in O, and $a \in t'$.

Each client or subcontractor consists of a set of *short-duration transactions*, together with some order defined on them. A subcontractor is (t, o, c) where:

- t is a set of short-duration transactions
- o is a user-defined order
- c is the user's notion of consistency.

2.3.4 Implementation Techniques of Nested CAD Transaction

Below we describe the concurrency control requirement at each of the transaction levels.

- Serializability of concurrent project transactions can be enforced at the database operation level by a two-phase locking algorithm.
- Cooperating transactions require atomicity of constituent short-duration transactions. A two-phase locking algorithm can enforce this.
- Subcontractor hierarchies require a multilevel concurrency control scheme, implemented at the database operation level. We can use two algorithms, a virtual timestamp ordering algorithm and a two-phase checkout algorithm to support the atomicity requirements of client/subcontractor hierarchies.
- Database operations must be executed in an atomic fashion. This must be enforced at the system operation level by guaranteeing serializability through a two-phase locking algorithm.

Now, we describe implementation techniques for the concurrency and recovery of nested CAD transactions.

2.3.5 Concurrency control for cooperating transactions

We can impose standard two-phase locking at the level of project transactions, so that a lock request by a transaction that belongs to one project transaction is rejected if it conflicts with a lock held by a different project transaction. A project transaction establishes its partition of the database by dynamically acquiring and releasing locks on the database. A project transaction locks a data item, not to access it directly, but to allow its constituent CAD transactions to access the data. Since a project transaction has many short-duration transactions executing concurrently, it is necessary to impose locking at the short-duration transaction level.

In our protocol, a CAD transaction whose lock request is denied, because another transaction belonging to the same project already holds a conflicting lock, has three options: (1) wait for the lock to be grantable, (2) be notified of the denial of the request and be allowed to continue, and (3) the short-duration transaction which issued the lock request on behalf of the CAD transaction can be rolled back. However, if the request is denied because another project holds a conflicting lock, the requesting transaction should not be made to wait, since the wait may be long. There are only two options: (1) notify of the conflict, and (2) roll back the short-duration transaction that issued the lock request.

2.3.6 Concurrency control for subcontractor transaction

In this subsection we present two protocols that may be used to implement the client-subcontractor transaction nesting.

A two-phase checkout algorithm

The protocol pertains to the case in which the client's partial ordering places no constraints on the point at which the "subroutine call" to the subcontractor appears within the execution of the client. A client or subcontractor transaction has associated with it a client space, a private space and a subcontractor space. The *private space* of a transaction is not shared with any other transaction. A transaction can read and update data in its private space. The *subcontractor space* is where a transaction places private data that a subtransaction may check out. A transaction has a unique subcontractor space. The *client space* of a transaction is the subcontractor space of its client, if there is one; otherwise, it is the public database.

A *checkout* is the moving of data by a transaction from its client space to its private space. The reverse of a checkout is a *checkin.* A *checkout enable* is the moving of data by a transaction from its private space to its subcontractor space. A *checkout disable* is the moving of data from a transaction's subcontractor space back to its private space.

We say that a client-subcontractor transaction observes the *two-phase checkout protocol* if it checks out data from its client space during one phase and checks them back in during the next phase, so that once it checks in any data, it cannot check out any more data. In the two-phase checkout protocol, a checkout is analogous to a lock request and a checkin is analogous to a lock release.

A virtual timestamp algorithm

This approach is suitable for situations in which there is a partial ordering in the client that constrains acceptable subcontractor executions. *Virtual timestamping* is described as follows:

When we begin a client-subcontractor transaction T, we assign to it a *start time* ST and a *duration* D. If that transaction consists of a sequence of short-duration transactions 1, 2, ..., n, we will assign to each short-duration transaction i a start time $st(i)$ and a duration $d(i)$. Then the following equations have to be satisfied.

$$ST \leq st(1) < st(2) < ... < st(n) < ST + D \tag{1}$$

$$st(i) + d(i) \leq st(i+1) \tag{2}$$

When a short-duration transaction i of T spawns a subcontractor T', it assigns to it a start time ST' and a duration D' such that

$$st(i) < ST' < ST' + D' < st(i+1) \tag{3}$$

The above is repeated recursively for the depth of the client-subcontractor hierarchy. Atomicity of subcontractors is guaranteed if Equations (1), (2), and (3) above are satisfied.

2.3.7 Recovery of nested transactions

The techniques most widely used for transactions recovery in database systems are shadowing and logging. However, CAD database systems must support multiple concurrent transactions, and the difficulty in supporting concurrent transactions using only the shadow mechanism leads us to the logging approach. In traditional database systems, crash recovery requires that all transactions active at the time of a crash be aborted. In the long-duration transaction model, this requirement is undesirable. In this technique/approach, if a long-duration transaction is active at the time of a crash, any active short-duration subtransactions of the long transaction are aborted. The results of all committed subtransactions are restored.

2.4 Semantic Pattern Approach

Korth's constraint-based model described above relaxes serializability to allow subtransactions at lower levels in the hierarchy to cooperate as long as each subtransaction preserves its consistency constraint. Although this approach weakens serializability, individual objects are shared only via two-phase locking. Hence, a transaction can never read an object while another transaction is modifying it [KS88]. Skarra and Zdonik propose a transaction model which exploits more the characteristics of design activities and maintains database consistency according to some semantic correctness criteria [SZ89].

2.4.1 Skarra's patterns and conflicts

This model supports a nested transaction model similar to the one in Korth's model. A *cooperative transaction* is a sequence of atomic operations that is determined dynamically as the system task progresses. Cooperative transactions work together forming a *transaction group* [FZ89]. A transaction group may have members that are cooperative transactions, as well as members that are transaction groups. As a result, it supports nested transaction groups. For example, consider the word processing subsystem. Each of the three design transactions, DISPLAY-DVP, PROCESS-DVP, and STORAGE-DVP starts up a cooperative transaction. They together form a transaction group.

Cooperative data sharing is localized in transaction groups. A transaction group dynamically interleaves its members' operations to generate a semantically consistent, concurrency history. The correctness specification is in the form of *patterns* and *conflicts*. A pattern describes an operation sequence that preserves consistency within or between objects of the same or different type. Conflicts are defined within the context of patterns and specify operations that are prohibited at specific points. Many patterns and conflicts may be defined for a particular task, each constraining a different aspect of the cooperative transaction interactions. For instance, consider the following pattern:

P1: $WRITE_{DISPLAY-DVP}$ (disp-proc-if-spec) $\rightarrow$ $READ_{PROCESS-DVP}$ (disp-proc-if-spec) specifies that the transaction PROCESS-DVP must read the interface specification, disp-proc-if-spec, that comes from the last write in the transaction DISPLAY-DVP.

A *history* for a transaction group is the partial order of the read and write operations executed by its cooperative transactions. A history produced by a transaction group is guaranteed to be correct by delaying or refusing operations that do not conform to the pattern specification. The *consistency set* of a history, H, consists of all patterns *required* by operations in H. Pattern P is required by operation O when P designates operations that must precede or follow O for semantic correctness. For example, the above pattern P1 is *required* by $READ_{PROCESS-DVP}$(disp-proc-if-spec), but not by $WRITE_{DISPLAY-DVP}$(disp-proc-if-spec).

For each operation, O, the transaction group adds the required patterns to the consistency set, S. It then tests O against patterns in S that are unsatisfied. O proceeds if it conflicts with none of their previous operations. If O completes successfully, it is appended to the history.

An operation, O1, in pattern P1, conflicts with a previous operation, O2, in P2, when it violates a condition set by O2, and an upcoming operation in P2 depends on the condition. All these conflicts are stored in conflict tables. A table is defined for each type with a row and column for each of the type's operations. A nonblank entry at (O1, O2) consists of operations whose effect, when preceded by O1, depends on whether O2 executes after O1.

2.4.2 Nodine's operation machine

A new, user-definable mechanism for expressing *patterns* and *conflicts* in a transaction group is proposed in [NZ90]. It utilizes *operation machines* which are actually finite state automata. The starting state represents the beginning of a pattern. Machine transitions represent operations on an object by some transactions. They are annotated with return values that are either "a" (accept) if the operation conforms to the pattern, "r" (refuse) if the operation conflicts, or "q" (queue) when the operation conflicts now, but may conform to the pattern if done later. The lack of a transition for an operation from some state indicates that the operation is not relevant to the pattern at that time. Therefore the pattern cannot cause the operation to be queued or refused. The final states of an operation machine indicate when its pattern is complete in the history. Figure 1 shows an example of an operation machine. The final state is denoted by a double circle.

Each transition in an operation machine is a tuple <M,op,O,P> where

M $\in \{any, m_i, \bar{m_i}\}$ is some member, where *any* is any transaction
m_i is an identifier for some member i,
$\bar{m_i}$ is any member except m_i.
op $\in \{r, w\}$ is an operation, where r is read and w is write.
O is the object identifier of the target object.
P $\in \{a, r, q\}$ is a return value, where a is accept, r is refuse, and q is queue.

This operation machine forces DISPLAY-DVP to read the interface specification, disp-proc-if-spec, after the last write before it can actually make the corresponding changes in the program, display.c. It also forces display.c to be current with disp-proc-if-spec before the task is complete.

In general, these operation machines put a general form on the allowable executions consistent with the transaction group's task.

Operation machines may be instantiated from *operation machine templates*. An operation machine template is defined in the same way as an operation machine, but the transition functions may have variables for the member and object identifiers. Figure 2 shows an example of an operation machine template.

This machine template is to enforce one type of cooperation. Before member M can write the object, it must have read it. Furthermore, if some other member writes the object, then member

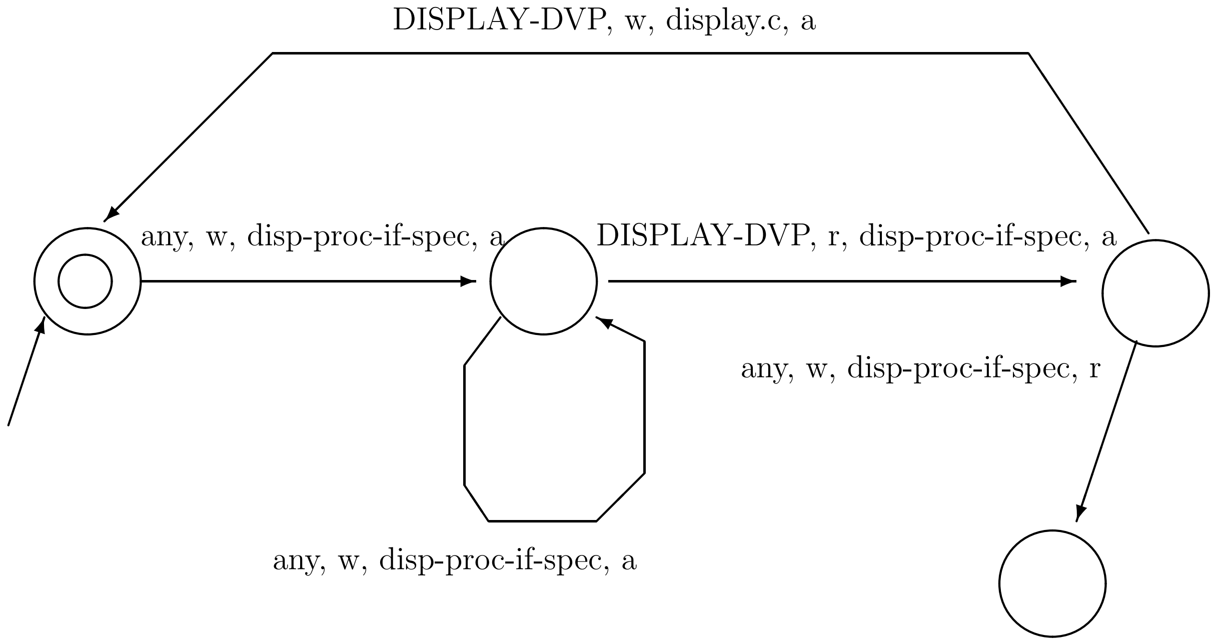

DISPLAY-DVP, w, display.c, a
any, w, disp-proc-if-spec, a
DISPLAY-DVP, r, disp-proc-if-spec, a
any, w, disp-proc-if-spec, r
any, w, disp-proc-if-spec, a

Figure 1: An operation machine

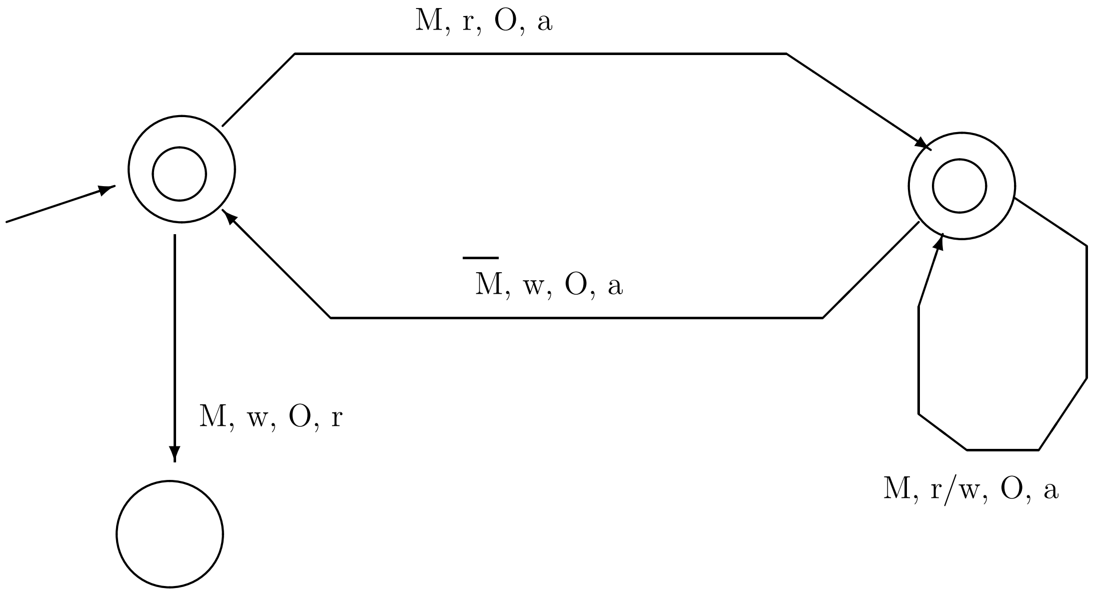


Figure 2: An operation machine template

M must read the written changes before it can write the object again. This machine prevents a member from overwriting another member's changes.

Each transaction group has one machine template for enforcing the underlying concurrency control mechanism for that group. The machine template with variable M for a single member identifier and O for a single object. For each <member,object> pair where the member is currently interacting with the object, there is an instantiation of the synchronization machine template with M bound to the *member* and O bound to *object*.

When an operation is requested by a member, it must be accepted by the operation machines before it can be executed. The object O specified in the request may be associated with many machines, each enforcing a different pattern for the members interacting with it. Each such machine may be relevant to the requested operation. Together, the machines enforce the overall correctness of the operations. An operation is guaranteed to be correct by ensuring that the operation causes no traversal of an arc whose predicate returns q (queue) or r (refuse). There may be several operation machines bound to the object O and member M specified in the operation request. For each such machine, the arc labeled either <M,op,O,P> or <S,op,O,P> (where M $\in$ S) from the current state in the operation machine defines how this operation participates in the associated pattern. There are four cases:

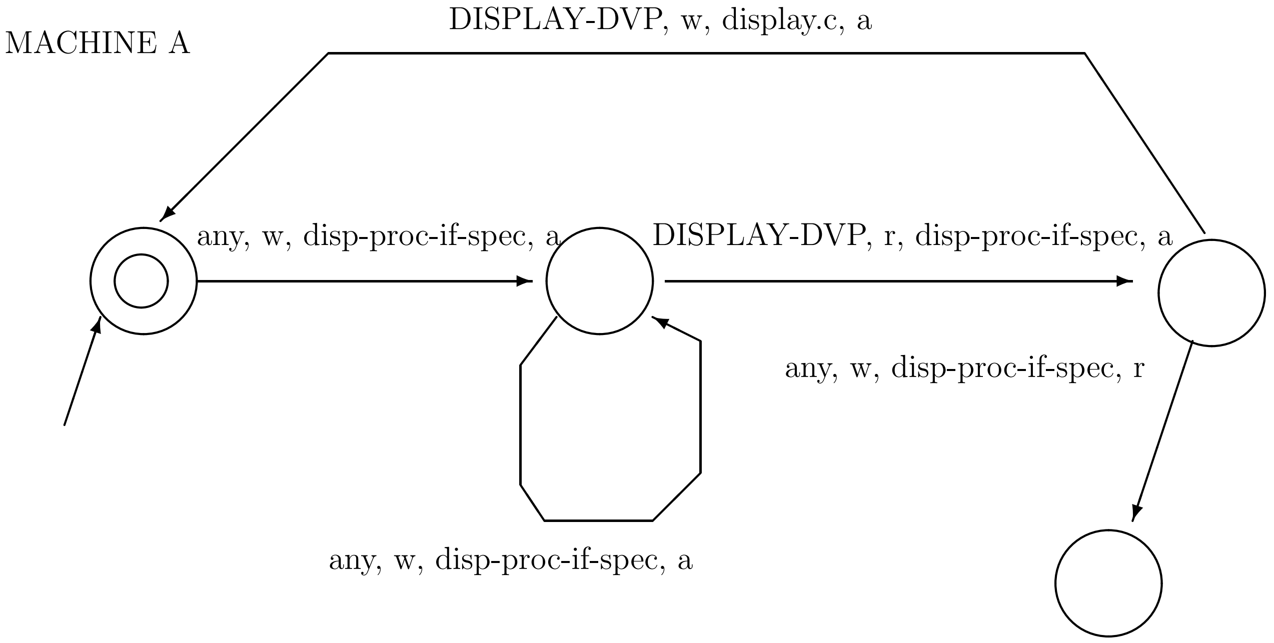
- P = a (accept). This operation is a correct continuation of this traversal at this time.
- P = q (queue). This operation does not correctly continue the traversal at this time, but may be accepted later.
- P = r (refuse). This operation cannot correctly continue the current traversal.
- There is no transition in this machine. This operation does not participate in this machine's pattern. This is an implicit accept.

If any machine returns r (refuse) for the requested operation, it is immediately refused. Otherwise, if any machine returns q (queue), the request is queued. If no machine returns r or q, the operation is accepted.

For example, assume that the object disp-proc-if-spec has two operation machines associated with it, as shown in Figure 3. The operation machine A is the same as the previous example, and the machine B is an instantiation of the operation machine template described above. Now the current state is the start state in both machines. The operation request <PROCESS-DVP, w, disp-proc-if-spec> would be refused, since the transition in machine B returns r. On the other hand, the operation request <PROCESS-DVP, r, disp-proc-if-spec> would be accepted, because it is not relevant to machine A and the transition in machine B returns a.

3 Data Approach

Models of semantic concurrency control which define concurrency properties on abstract data types according to the semantics of the type and its operations belong to *data approaches*, such as [Sch84] and [Her86]. They use the relationship between operations defined on a data type to optimize concurrency.


MACHINE A
DISPLAY-DVP, w, display.c, a
any, w, disp-proc-if-spec, a
DISPLAY-DVP, r, disp-proc-if-spec, a
any, w, disp-proc-if-spec, r
any, w, disp-proc-if-spec, a

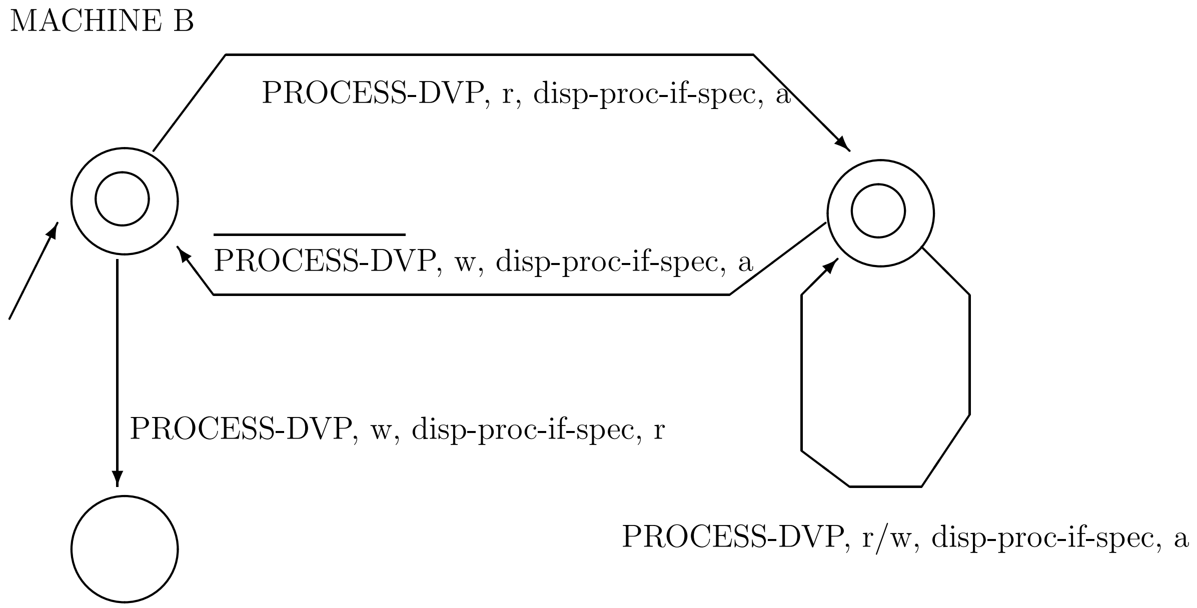

MACHINE B
PROCESS-DVP, r, disp-proc-if-spec, a
$\overline{\text{PROCESS-DVP}}$, w, disp-proc-if-spec, a
PROCESS-DVP, w, disp-proc-if-spec, r
PROCESS-DVP, r/w, disp-proc-if-spec, a

Figure 3: Operation machines for disp-proc-if-spec

The models of the data approach synchronize transactions under semantic serializability. A conflict relation is defined over the operations on each abstract data type, such that two operations conflict when either the semantic effect or the validity of a history is dependent on the order of their execution. A history is semantically equivalent to a serial history when conflicting operations appear in the same relative order in both. Synchronization algorithms use conflict-based locking schemes to generate histories that are serializable in the commit order of the concurrent transactions. Conflict-based locking is a generalization of traditional locking protocols that are based on Read/Write semantics.

3.1 Schwarz's model

In the database literature, transactions are defined as arbitrary collections of database operations bracketed by two markers: *BeginTransaction* and *EndTransaction.* A transaction that completes successfully commits; an incomplete transaction can terminate unsuccessfully at any time by aborting. Transactions have the following special properties:

1. Either all or none of a transaction's operations are performed. This property is usually called *failure atomicity.*

2. If a transaction completes successfully, the effects of its operations will never subsequently be lost. This property is usually called *permanence.*

3. If a transaction aborts, no other transactions will be forced to abort as a consequence. *Cascading aborts* are not permitted.

4. If several transactions execute concurrently, they affect database as if they were executed serially in some order. This property is usually called *serializability.*

Schwarz's model [Sch84] for using transactions in distributed systems differs from traditional model in several ways. The most important difference is that [Sch84] incorporates the concept of an *abstract data type.* That is, information is stored in typed *objects* and manipulated only by *operations* that are specific to a particular object type. The users of a type are give a *specification* that describes the effect of each operation on the stored data, and new abstract types can be implemented using exist ones. The details of how objects are represented and how the operations are carried out are known only to a type's implementor. Another difference between Schwarz's model and the traditional transaction model is that one does not necessarily require that transactions appear to execute serially.

3.1.1 Intertype Synchronization

[Sch84] extends the traditional transaction model by redefining a transaction as a sequence of typed operations on objects that are instances of shared abstract types. Semantic knowledge about individual types is used in the model to develop synchronization and recovery strategies that allow more concurrency. Some of the strategies produce nonserializable results. The semantics include information about the typed operations, as well as the arguments and the results of operations.

The synchronization protocol is made up of several components:

- the execution of a group of concurrent transactions is characterized by an ordering property. The property may be serializability, or it may be a weaker correctness condition.
- concurrency specifications are defined on each abstract data type that satisfy particular ordering properties of transactions that access instances of the type. Concurrency control is determined locally by a type for its instances.
- the system provides atomicity of transactions.

In particular, the synchronization protocol embodies the following requirements for operations on shared abstract types:

- An operation cannot be permitted to observe or manipulate the state of an object during the invocation of another operation on the project. This requirement prevents the propagation of inconsistent data.
- The ordering property of a group of transactions must be preserved in interleaving the transactions' operations at each object. Moreover, the property must be preserved consistently across all objects shared by the transactions. This requires local synchronization of an object to cooperate with that of other objects to preserve the ordering property.
- Transactions cannot view data that might change if another transaction were to abort. This requirement prevents cascading aborts.

3.1.2 Dependencies

Concurrency control in the model uses the notion of dependencies between transactions. A dependency forms between two transactions when they both access the same object. For example, assume T1 and T2 are two concurrently executing transactions. T2 is dependent on T1 if and only if T1 performs an operation on an object and then T2 performs an operation on the same object. A schedule resulting from the interleaving of T1 and T2 is serializable when the dependencies that form during its execution are acyclic. A cycle in the dependency graph would mean, for example, that some operation in T2 occurs after some operation in T1 that occurs after some other operation in T2. Thus, the interleaved schedule could not be serialized such that either T1 preceded T2 or vice versa.

The dependencies described above assume no semantic information about the operations or the object. No matter what operations are invoked, edges are added to the dependency graph when transactions touch the same object. However, the use of semantics may allow some dependencies to be ignored (that is, we need not add an edge to the graph), without violating the correctness of the interleaved history. For example, if a cycle in the graph results from T1 and T2 both reading the same object, the schedule is still serializable. The dependency resulting from two transactions reading the same data is *insignificant* with regard to serializability.

The concurrency semantics for an object are defined as a dependency specification with the object's abstract data type. A specification indicates, for all possible pairs of operations defined on the type, which of the resulting dependencies between transactions are significant with regard to a given correctness criterion (for example, serializability). If a specification is not given with a type, dependencies form between transactions whenever they access the same instance of the type, regardless of the operations invoked.

As an example, consider the type Book on which the operations Read and Write are defined. The following is the complete set of dependencies that form when transactions read or write books, with the notation

$$D_i \qquad T_j : X < T_k : Y$$

signifying that the ith kind of dependency forms when transaction T_j invokes the operation X, and then T_k invokes Y on the same book; T_k becomes dependent on T_j.

$$\begin{array}{ll} D1 & T_j : Read < T_k : Read \\ D2 & T_j : Read < T_k : Write \\ D3 & T_j : Write < T_k : Read \\ D4 & T_j : Write < T_k : Write \end{array}$$

In this example, there are four kinds of dependencies that can form when transactions access books. However, only the last three are significant in a concurrency control mechanism that provides serializability. D1 dependencies can be ignored because of the semantics of the Read operation. Thus, the dependency specification for Book that provides serializability is composed of D2, D3 and D4 dependencies. The concurrency control mechanism need only keep track of those dependencies in the dependency specification (that is, the graph contains edges only for D2, D3, and D4 dependencies).

3.1.3 Cooperative Types

The synchronization protocol generates histories whose correctness is determined by the ordering property of the concurrent transactions. The sequence of operations in a global, interleaved history that satisfies the property, however, is determined by concurrency specifications defined with the abstract data types whose instances are manipulated by the transactions. That is, the operation sequence allowed at a given object is determined locally by its type, rather than by a global specification. Hence, the global history, as the aggregate of the objects' local histories, is determined by local specifications. For example, assume that the correctness criterion for a history of concurrent transactions is serializability, and the types manipulated by the transactions define specifications that satisfy serializability. Operations invoked at each object will be scheduled according to the specification of its type, and the resulting local sequence will be serializable within the context of the object.

When the transactions touch objects of different types, however, the equivalent serial ordering(s) of local schedules must be mutually consistent in order for global serializability to be satisfied. If transactions T1 and T2 are both modifying a Queue object and a Directory object, then in both objects either T1 precedes T2 or T2 precedes T1. Any other ordering is not globally serializable, even though the schedule at each object may be locally serializable. The model accommodates synchronization across types, such as Queue and Directory, by designating them as *cooperative types*. The dependency specification for a group of cooperative types consists of the combined specifications of the individual types in the group. The same mechanism then applies uniformly to objects of the same or of a different type. Dependencies that form between transactions accessing one of the cooperative types are added to those between transactions accessing another of the types. An absence of cycles in the resulting group of dependencies indicates a serializable schedule when each of the participating types defines a dependency specification that guarantees serializability.

3.1.4 Concurrency Control

The implementation of the concurrency control scheme just described does not actually allow dependencies to form. Assume a transaction, T2, is allowed to read data written by an uncompleted transaction, T1. T1 could write to the data again before committing, or it could abort. Thus, T2 may receive data that are inconsistent or at least different from the committed data. Moreover, the data may be passed to other transactions that are dependent on T2. Cascading aborts can result. Instead, the implementation includes a locking algorithm that suspends T2 rather than allowing a dependency on T1 to be formed.

The locking algorithm uses the dependency specification of a type as the definition of conflict between the operations of type. Two operations on a type conflict with each other if a significant dependency would form between transactions invoking them. In the Book example, Read and Write operations both conflict with Write because of D2, D3, and D4 dependencies. Read operations do not conflict because a D1 dependency is not significant. The locking algorithm uses the operation conflicts to introduce delays in transaction processing. If transaction T2 invokes an operation O on an object that conflicts with an operation previously invoked by T1, the execution of O is delayed and T2 is suspended until T1 commits. The delays induce a serialization order on the schedule; in particular, the resulting order of the equivalent serial schedule is consistent with the commit order of the transactions.

3.1.5 FIFO Queue Semantics

The viewing of operations as simply Reads or Writes, rather than exploiting more detailed semantic information, may result in concurrency limitations that are too strict for a given type. Instead, the type designer can make use of operation semantics, including operation arguments and results, in the dependency specification to further increase concurrency. [Sch84] describes the FIFO Queue as an example; it has two operations.

1. QEnter(q,p): Adds an entry p to the end of the Queue object q. The undo operation for QEnter removes the entry.

2. QRemove(q): Removes the entry at the head of q and returns the value of p contained therein. If q is empty, the operation blocks and waits until q becomes nonempty. The undo operation for QRemove restores the entry to the head of q.

In addition, queues have numerous restrictions that must be maintained by the transaction scheduler. For instance, entries made by the same transaction must appear in the queue as a contiguous block. Also, entries made by a transaction cannot be observed by other transactions until the entering transaction commits.

The dependency specification for the Queue type makes use of the arguments passed to the operations and the results that they can return. It is assumed that each entry in the queue can be uniquely identified; for the purposes of the following specification, p_j is considered to be a different queue entry from p_k. The complete set of dependencies that form when transactions access queues is as follows, with the notation E(p_j) signifying that the transaction entered p_j in the queue and R(p_k) signifying that the transaction removed p_k from the queue:

$D1$	$T_m : E(p_j) < T_n : E(p_k)$
$D2$	$T_m : E(p_j) < T_n : R(p_k)$
$D3$	$T_m : E(p_j) < T_n : R(p_k)$
$D4$	$T_m : R(p_j) < T_n : E(p_k)$
$D5$	$T_m : R(p_j) < T_n : R(p_k)$

The dependency specification for queues that guarantees serializability consists of D1, D3, and D5 dependencies. D2 dependencies are insignificant, since the ordering of concurrent transactions performing the operations cannot be detected by the transactions themselves or by any future transactions. Similarly, D4 as the inverse of D2, is insignificant. Casting the operations as Reads and Writes would unnecessarily limit concurrency. QEnter would be cast as a Write operation, and QRemove would be cast as a Read followed by a Write. Consequently, both D2 and D4 dependencies would have to be included in the specification in order to guarantee serializability.

3.1.6 Weakly-FIFO Queue Semantics

For some abstract data types, serializability may be too restrictive as a correctness criterion. In these cases, a weaker ordering property may be more appropriate by virtue of the higher levels of concurrency achieved without a sacrifice in type-specific consistency. A weaker ordering property is obtained by including fewer of the complete set of dependencies that can form between transactions in the type's dependency specification.

For example, Queues are frequently used as producer/consumer buffers. The exact ordering of entries on the queue is not particularly important, provided an entry reaches the head of the queue at approximately the same time as other entries of the same vintage; an entry should not languish in the queue forever. Queues of this nature are *weakly-FIFO*. The complete set of dependencies is the same as that defined for the FIFO Queue type; however, the dependency specification for the weakly-FIFO Queue type includes only D3 dependencies. Entry order is not strictly serializable, but cascading aborts are avoided, since an entry is not visible to other transactions until it has committed. Moreover, the semantics of the data type are supported by the concurrency protocol, so that weakly-FIFO queues are guaranteed to be semantically consistent.

3.2 Herlihy and Weihl's model

Herlihy and Weihl describe a model [Her86] [HW88] for concurrency control and recovery that satisfies hybrid atomicity and uses a serial dependency relation as the basis for defining operation conflicts. Informally, conflict is defined in terms of invalidation. Two concurrent transactions execute no conflicting operations and can be serialized in either order when neither transaction invalidates the other. The notion of invalidation as a basis for conflict is weaker than commutativity, which requires that both serializations define equivalent states.

A dependency relation can be defined over the operations of each type, such that an operation, O1, depends on any other operation, O2, that can invalidate it by appearing earlier in a sequence. That is, if there exist operation sequences H1 and H2 such that H1 O2 O1 is legal sequence, but H1 O2 H2 O1 is not, then O2 invalidates O1, and O1 depends on O2. For example, the Account type has Deposit and Withdraw operations. Withdrawals return successfully if accounts have sufficient funds; otherwise, they return an error. An unsuccessful Withdraw can be invalidated by prior Deposit operations, but successful Withdraws cannot. Thus, unsuccessful Withdraw operations depend on (i.e., conflict with) Deposits, but successful ones do not.

The protocol described in [HW88] manages concurrent access to an object with a timestamping and locking scheme that is based on the dependency-based conflict relation defined by the object's type. The protocol maintains several components for each object:

- An intentions list for each transaction consisting of the sequence of operations that are to be applied to the object if the transaction commits.
- The object's committed state, which reflects the effects of committed transactions. The committed state may be thought of as the intentions lists for the committed transactions, arranged in timestamp order.
- A set lock that associates each operation with the set of active transactions that have executed that operation. Locks are related by a symmetric conflict relation that can take arguments and results into account.

The result of an operation invoked by transaction T on an object is obtained by executing the operation in the state that is derived by appending the intention list of T to the committed state of the object. The result is computed before the new operation is actually appended to T's intentions list, since T must first obtain a lock for the operation, and the model allows operation arguments and results to be used in defining conflict relations. If a conflict exists, the lock request is refused and the result is discarded. The invocation is later retried (and may at that time return a different result). If the lock is granted, the operation is appended to the intentions list and the result is returned. When a transaction commits, its intentions list is merged into the committed state in timestamp order.

[HW88] shows that protocol is less restrictive and can achieve at least as much concurrency as commutativity-based protocols. Further, lock conflict relations induced by dependency may be weaker than or incomparable to those defined by commutativity. The concurrency advantage is especially apparent for operations that do not return the state of the object as part of their result. The definition of commutativity used in the comparison is forward commutativity.

4 Conclusion

The discovery and improvement of the shortcomings of conventional database techniques have made database management enter a new phase – **object-oriented database systems**. The main difference between an object-oriented database system and a non-object-oriented database system is that an object-oriented database system can directly support the needs of the applications that create and manage objects that have the object-oriented semantics. Although many of the techniques from traditional transaction processing can be carried over to object-oriented databases, the model of transactions supported in conventional database systems is inappropriate for long-duration transactions necessary in interactive, cooperative design environments. New semantics in object-oriented databases need to be used to improve concurrency and to express correctness criteria other than serializability. New models of transactions suitable for long-duration transactions need to be explored in object-oriented database systems.

In our report, we introduced two classes of semantics approaches for transactions in object-oriented database systems – one is the transaction approach, and the other is the data approach. Models of the *transaction approach* define concurrency properties on transactions according to the semantics of the transactions and the data they manipulate, while models of the *data approach* define concurrency properties on abstract data types according to the semantics of the type and its operations. In the transaction approach, the interleaving of concurrent transactions is explicitly constrained by specifications on transactions, while in the data approach, interleavings are implicitly constrained by operation conflicts defined on abstract data types. In the data approach, an object offers a uniform, concurrent behaviour, regardless of the semantics of applications using it. It is a more modular approach and allows decentralized concurrency control. In contrast, the transaction approach requires control that is centralized with respect to each group of concurrent transactions, and operation semantics have to be reconsidered in the context of each new transaction. In the transaction approach, it is easy for application semantics to be added to the concurrency control. In the data approach, semantics contributed by applications are added as new operations on data types. Addition of application semantics to types may be viewed as a violation of modularity and the addition of operations to a type may be complicated by problems associated with schema changes.

In the transaction approach, we described three models suitable for long-duration transactions. The first model categorizes the transactions into different types through their semantics information. Each transaction is divided into atomic steps. The second model gives a formal model of nested transactions appropriate for CAD environments. The third model proposes a similar nested model of transactions. However, it provides a more expressive construct to specify the concurrency control through augmented finite state automata.

In the data approach, we presented two models. The first model extends the traditional transaction model by redefining a transaction as a sequence of typed operations on objects that are instances of shared abstract types. The second one describes a model for concurrency control and recovery that satisfies hybrid atomicity and uses a serial dependency relation as the basis for defining operation conflicts.

References

[BK85] W. Bancilhon, F. Kim and H.F. Korth. A model of CAD transactions. *Proceedings of 11th International Conference on Very Large Data Bases*, pages 25–33, 1985.

[FZ89] M. Fernandez and S. Zdonik. Transaction groups: A model for controlling cooperative transactions. In *3rd International Workshop On Persistent Object Systems*, pages 341–350, 1989.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[Her86] M. Herlihy. Optimistic concurrency control for abstract data types. In *Proceedings of the Fifth ACM Symposium on the Principles of Distributed Computing*, August 1986.

[HW88] M. Herlihy and W. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1988.

[KB88] W. Korth, H.F Kim and F. Bancilhon. On long-duration CAD transactions. *Information Sciences*, 46:73–107, 1988.

[KS88] H.F. Korth and G.D. Speegle. Formal model of correctness without serializability. In *ACM SIGMOD, Proceedings of International Conference on Management of Data 88*, pages 379–386, 1988.

[NZ90] A.H. Nodine, M.H. Skarra and S.B. Zdonik. Synchronization and recovery in cooperative transactions. In *The Fourth International Workshop on Persistent Object Systems*, pages 329–342, 1990.

[Sch84] P.M. Schwarz. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[SZ89] A. Skarra and S. Zdonik. Concurrency control and object-oriented databases. In *Research Directions in Object-Oriented Programming*, pages 395–421. MIT Press, Cambridge Mass, 1989.