Data and Time Abstraction Techniques for Analyzing Multilevel Concurrent Systems

TOSHIMI MINOURA, MEMBER, IEEE, AND S. SITHARAMA IYENGAR

Abstract—***Data abstraction*** **has long been recognized as an important tool for designing and analyzing a complex software system. In a concurrent system an execution time of an operation is as important as its functional effect. In this paper we argue that the design and analysis of a concurrent system can be made simpler and more intuitive if execution times of abstract operations are arbitrarily but systematically defined. This technique (*time abstraction*) is complementary to data abstraction and is more effective when used in combination with data abstraction. As examples, we analyze a bounded-buffer monitor and a multilevel concurrency control scheme of a database system by using data and time abstraction.**

Index Terms—**Access path, bounded buffer, B-tree, data abstraction, multilevel concurrency control, multilevel concurrent system, phantom problem, time abstraction.**

I. Introduction

ABSTRACTION is probably one of the most versatile and powerful mechanisms employed by human beings coping with complex phenomena. In a sense, abstraction is to look at things in a way we like them to be inasmuch as they behave consistently within our concern. In the programming field, *data abstraction* has long been recognized as an important means to make the design, implementation and analysis of a complex software system manageable. The first theoretical basis of data abstraction was presented in [22] within the context of correct program translation. Later, Hoare discussed the problem more explicitly and more precisely [11]. Since then work on data abstraction for sequential programs has been studied in detail [8]–[10], [18], [19], [30].

However, as stated in [18], most of the techniques developed for sequential programs cannot be directly applied to concurrent systems. In the case of a sequential program, we can simply consider that abstract operations are executed in the order in which collections of concrete operations (e.g., subroutines), each of which implements an abstract operation, are executed. In a concurrent system, collections of concrete operations implementing different abstract operations may be executed concurrently. Thus, it may not be simple to decide the execution order of these abstract operations. However, we show that it is often possible to specify explicitly the execution times for abstract operations and still to maintain a consistent view. We refer to this technique as *time abstraction*. The main objective of this paper is to emphasize the usefulness of time abstraction techniques, whose importance has not been fully recognized.

An important basis of time abstraction is the concept of *atomicity* [5], [21]. A collection of operations that are executed over a certain period of time, is *atomic*, if the net effects of those operations are as if they were executed instantaneously at some point in time. Such a set of operations is usually called a *transaction*. In this paper we proceed further to assign an artificial execution time to an abstract operation represented by a transaction. Note that for a concurrent system, the execution time of an operation is as important as how it transforms data. The execution times of abstract operations must be so selected that the analysis of the resultant execution becomes easy.

Once transactions implementing abstract operations at some level of a multilevel concurrent system are made atomic, those abstract operations can be assumed to take place instantaneously when they are invoked from higher levels. This fact can make the design, implementation, and analysis of a complex concurrent software system significantly easier. Further, an execution of a multilevel concurrent system can be controlled at each level; abstract operations at each level can be synchronized based on their hypothetical execution times.

The first step taken in designing a complex software system is often to conceive an abstract system whose executions satisfy the users' requirements. This step is especially important for a concurrent software system because such an abstract system may be the sole criterion for correct system implementation. Abstract operations that perform well-defined functions instantaneously make the specification of a system simpler.

Data abstraction in a concurrent system is addressed in [13], [26] by using other techniques, which are either more indirect or more complex than the one discussed in this paper. A closely related subject, i.e., *control (procedural) abstraction*, is discussed in [14], [17] as "reduction." Recent investigations on multilayered transaction processing systems can be found in [3], [24], [25].

In Section II, we present a model of a multilevel concurrent system as a framework for our methodology. In Section III, we analyze a bounded-buffer monitor by using a simple data and time abstraction technique. Section IV addresses the multilevel concurrency control problem for

Manuscript received September 9, 1986; revised January 2, 1987.

T. Minoura is with the Department of Computer Science, Oregon State University, Corvallis, OR 97331.

S. S. Iyengar is with the Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803.

IEEE Log Number 8824602.

a database system; the example is intended to show the usefulness of time abstraction when used in conjunction with data abstraction. Section V concludes this paper.

II. Multilevel Concurrent System Model

In this section we describe a model of a *multilevel concurrent system*.[1] At each level of a multilevel system, a set of (*abstract*) *operations* that operate on (*abstract*) *data* are supported. Abstract data at one level, however, may be imaginary entities that are represented by lower level data, which again may be abstract data. When the data at level L is represented by the data at level L', operations at level L must be implemented by the operations at level L'.

We characterize a *level* of a multilevel system by a triple (D, s_0, O), where D is the set of (*abstract*) *objects* supported at this level, s_0 specifies the *initial values* of the objects in D, and O is the set of *operations* provided at this level. An object contains a *value*. A *system state* is a function that designates the values of the objects in D. Associated with each operation o is a partial function $f_o: S \times I_o \rightarrow S \times U_o$, where S is the set of all possible system states, I_o the set of all possible *input lists* for o, and U_o the set of all possible *output lists* produced by o. The fact $f_o(s, i_o) = (s', u_o)$ means that if operation o occurs with system state s and with list i_o of input values, then the new system state will be s', and list u_o of output values will be produced. We assume that a finite description of f_o is provided for each operation o, and that given s and i_o, s' and u_o can be obtained as $(s', u_o) = f_o(s, i_o)$. The description of f_o may be informal.

We represent a *basic execution* (*b-execution*) at level (D, s_0, O) by $e = (OI, t, i, u)$, where OI is the set of *operation instances* in e, $t: OI \rightarrow R$ defines the *occurrence times* of the operation instances in OI, i, which is called the *system input-list*, is the list of input values provided for e, and u, which is called the *system output-list*, is the list of output values produced by e. We denote by $op(a)$ the operation one of whose instances is a. If a is in OI, $op(a)$ is in O. Each operation instance a is assumed to occur instantaneously at time $t(a)$. (A justification of this assumption is the main theme of this paper.) We assume that the number of input values consumed and the number of output values produced by each instance of an operation are uniquely defined by the system state when the operation is activated and the input values provided for it. The system input- and output-lists, i and u, are obtained by concatenating the input and output lists, respectively, of the operation instances according to their execution order. Therefore, i uniquely designates the input values provided for each operation instance, and u the output values generated by it.

Consider a particular execution that is taking place, and let $e = (OI, t, i, u)$ be the *current* b-execution, which is the b-execution that occurred so far. Further, let s be the current system state, which is the system state produced by the last operation instance in OI or is s_0 if no operation instances have occurred yet. Assume that operation instance a occurs at this point, which is time t_a, with its input list i_a. Then a will update the system state from s to s' and produce the list u_a of output values as specified by $(s', u_a) = f_{op(a)}(s, i_a)$. The current b-execution will become $e' = (OI \cup \{a\}, t \cup \{(a, t_a)\}, i @ i_a, u @ u_a)$, where "@" is the list-concatenation operator.

It is often beneficial to treat a group of operation instances (e.g., the set of operation instances generated by a procedure) as a unit. An execution in which operation instances are grouped will be called a *molecular execution* (*m-execution*).

If procedures are executed one at a time, they can be considered as (macro) operations. An m-execution based on procedure activations can be defined like a b-execution based on operation instances. The fact that if a procedure p is activated with a system state s and an input list i_p, it will update the system state to s' and produce the output list u_p can be designated as $(s', u_p) = f_p(s, i_p)$, where f_p is a function specifying the effect of p. We represent a *serial m-execution* of a set P of procedures as $e_{sm} = (PI, t, i, u)$, where PI is a set of activations of the procedures in P, t specifies the occurrence time of each procedure activation in PI, i designates the system input-list obtained by concatenating the input lists for the procedure activations in PI, and u the system output-list obtained from the output lists produced by the procedure activations in PI.

We now want to allow concurrent executions of the procedures. Since allowing indiscriminate interleaving of the operation instances generated by different procedures is likely to lead to difficulties in designing such procedures, we assume that the set of operation instances generated by an activation of each procedure can be partitioned into *transactions*, each of which is *atomic*. A transaction consisting of a set of operation instances is atomic if the net effects of those operation instances are as if they occur instantaneously at some point in time. Transactions can be executed concurrently as long as their effects are atomic. The operation instances generated by an activation of one procedure may belong to one transaction, or they may be divided into multiple transactions.

We represent a *concurrent m-execution* of a set P of procedures as $e_{cm} = (PI, T, t, i, u)$, where PI is the set of procedure activations, T the set of the transactions created by the procedure activations in PI, t specifies the *effective occurrence time* of each transaction in T, i designates the system input-list obtained by concatenating the input lists for the procedure activations in PI, and u the system output-list obtained from the output lists produced by the procedure activations in PI. Note that each transaction T_i in T is supposed to occur instantaneously at time $t(T_i)$. The effective occurrence time of each transaction can be defined in different ways according to the concurrency control scheme applied to the transactions. When two-phase locking [5] is used, the *lock-point* [4], [27] of each transaction can be used as its effective occurrence

[1] The model in this section extends those in [3], [23], [24].

time. When timestamping [28] is used, the timestamp value of a transaction can be used as its effective occurrence time.

In a multilevel system, operations at each level, except for the lowest level, are implemented by the procedures that use operations of lower levels. This mechanism is well-understood for sequential programs [11]. Our aim is to introduce concurrency in the executions of the procedures that implement higher level operations. The key feature of our strategy is to let the effect of each higher level operation take place instantaneously even though the procedure that implements the operation may be executed concurrently with other procedures. Therefore a programmer using operations at each level can assume that the effects of the operations at that level are atomic.

We now state our methodology for the design and analysis of a multilevel concurrent system. Fig. 1 shows our methodology graphically. Assume that level $L = (D, s_0, O)$ must be supported by its immediate lower level $L' = (D', s_0', O')$.

A1) Data and Operation Abstraction: We let each (meaningful) system state s of level L be represented by a system state s' of level L' as specified by $s = dabs(s')$, where *dabs* is a *data abstraction function*. In particular, it must be that $s_0 = dabs(s_0')$. Further, the following condition must hold for each operation o in O implemented by a procedure $prog(o)$ using operations in O'. If for states s_1 at level L and s_1' at level L' $s_1 = dabs(s_1')$, $f_o(s_1, i_o) = (s_2, u_o)$ and $f_{prog(o)}(s_1', i_o) = (s_2', u_o')$, then it must be that $s_2 = dabs(s_2')$ and $u_o = u_o'$. When an operation instance a of an operation o at level L is requested to occur with input list i_o, $prog(o)$ as its parameter list, and it is executed generating operation instances of level L'. When $prog(o)$ terminates, it produces the output list u_o, which can be regarded as produced by a itself. Procedures that are executed on behalf of the operations at level L are called *foreground procedures*. Further, procedures that consume empty input lists and produce empty output lists and that do not affect the abstract system state at level L may be executed. Such procedures will be called *background procedures*. Garbage collection and data-structure reorganization, for example, can be performed by background procedures. If a background procedure updates the system state at level L' from s_1' to s_2', then it must be that $dabs(s_1') = dabs(s_2')$. Now, a b-execution $e = (OI, t, i, u)$ at level L can be supported by serial m-execution $e_{sm}' = (PIF \cup PIB, t', i, u)$ at level L', where *PIF* is the set of foreground-procedure activations caused by the operation instances in *OI*, *PIB* is the set of activations of the background procedures, and t' specifies the occurrence times of the procedure activations in $PIF \cup PIB$. If a foreground-procedure activation p in *PIF* occurs responding to an operation instance a in *OI*, $t(a) = t'(p)$.

A2) Parallelization with Time Abstraction: We now introduce concurrency into the executions of the procedures at level L'. A concurrent m-execution is designated as $e_{cm}' = (PIF \cup PIB, TP \cup TA, t'', i, u)$. Operation instances at level L' must be partitionable into the set $TP \cup TA$ of transactions, where *TP* is the set of *principal* transactions, and TA the set of *auxiliary* transactions. Function t'' designates the effective occurrence time of each transaction in $TP \cup TA$. For each operation instance a at level L, there must exist exactly one principal transaction T_a in *TP*. Assume that for states s_1 at level L and s_1' at level L', $s_1 = dabs(s_1')$, that a transforms s_1 to s_2, and that T_a transforms s_1' to s_2'. It then must be that $s_2 = dabs(s_2')$. If an auxiliary transaction transforms the system state at level L' from s_1' to s_2', then it must be that $dabs(s_1') = dabs(s_2')$. An operation instance a associated with a principal transaction T_a should be regarded to take place when T_a is supposed to occur instantaneously; i.e., $t(a) = t''(T_a)$.

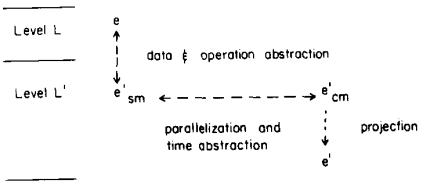


Fig. 1. Multilevel concurrent system.

A3) Projection: Responding to a b-execution $e = (OI, t, i, u)$ a level L, *b*-execution $e' = (OI', t''', i', u')$ is eventually generated at level L', since a concurrent m-execution at level L' is performed by operation instances at level L'.

Although b-execution e at level L is realized by b-execution e' at level L', t is defined by t''', and not the other way around; actually, t''' defines t'', and t'' defines t. Also, note that i' is defined from i and u', and that u is defined from u'. This mechanism can be best compared to the parameter binding mechanism used by Prolog programs.

III. Bounded-Buffer

In this section we analyze a simple concurrent program following the methodology described in Section II. We hope that our analysis clarifies the way time abstraction can be used in combination with data abstraction.

Fig. 2 shows a *bounded-buffer* mechanism designed as a monitor [12], [13], [26]. In Fig. 2, symbol "$+_N$" denotes the modulo N addition operator, and symbol "@" the concatenation operator. We define the data abstraction function *dabs* specifying the state of abstract data structure *buffer* as follows:

dabs(*COUNT*,*BUFFER*,*HEAD*,*TAIL*) =
  if *COUNT* = 0 then empty
  else *BUFFER*[*HEAD*] @ *BUFFER*[*HEAD*+1]
  @ . . . @ *BUFFER*[*TAIL*].

The bounded-buffer monitor supports two kinds of abstract operations (*append*(x) and *remove*(x)) on *buffer*. Let us denote the number of messages in *buffer* by $|buffer|$. When an abstract operation *append*(x) is issued, message x is appended to *buffer* as its last element if $|buffer| < N$. If $|buffer| = N$, the operation *append*(x) is suspended until the condition $|buffer| < N$ becomes true. When an abstract operation *remove*(x) is

```
BOUNDED-BUFFER: monitor

begin

  COUNT:      0 .. N;
  BUFFER:     array[0..N-1] of messages;
  HEAD:       0 .. N-1;
  TAIL:       0 .. N-1;
  NONFULL:    condition;
  NONEMPTY:   condition;

 procedure APPEND(x: message)
  begin
        {buffer = b, x = m}
ap1:    if COUNT = N then NONFULL.wait;
        {buffer = b, x = m}
        ----------
        {buffer = b, x = m}
ap2:    COUNT := COUNT + 1;
        TAIL := TAIL +N 1;
        BUFFER[TAIL] := x;
        NONEMPTY.signal;
        return
        {buffer = b @ m}
  end;

 procedure REMOVE(result x: message)
  begin
        {buffer = b}
rm1:    if COUNT = 0 then NONEMPTY.wait;
        {buffer = b}
        ----------
        {buffer = m @ b}
rm2:    COUNT := COUNT - 1;
        x := BUFFER[HEAD];
        HEAD := HEAD +N 1;
        NONFULL.signal;
        return;
        {x = m, buffer = b}
  end;

 COUNT := 0; HEAD := 0; TAIL := N-1

end
```

Fig. 2. Bounded-buffer monitor.

issued, the message at the head of *buffer* is returned as x. If such a message does not exist in *buffer*, the operation *remove*(x) is blocked until such a message becomes available.

A monitor consists of blocks of statements each of which is executed *indivisibly*, and hence an execution of each block can be regarded as a transaction. In Fig. 2, the effect of each block of statements is indicated by the *pre-* and *postconditions* attached to the block. The predicates in each $\{\cdots\}$ show the state of the abstract data (primarily, *buffer*).

This bounded-buffer monitor can be modeled as level L' and (D', s_0', O'), where $D' = \{COUNT, BUFFER, HEAD, TAIL\}$ and $\{s_0'(COUNT) = 0, s_0'(HEAD) = 0, s_0'(TAIL) = N - 1\}$. We do not specify O' explicitly since it is too detailed. A concurrent m-execution at level L' can be specified as $e'_{cm} = (PIF, TP \cup TA, t'', i, u)$, where PIF is a set of activations of procedures $APPEND(x)$ and $REMOVE(x)$, TP is the set of principal transactions caused by the executions of block ap_2 of $APPEND(x)$ and block rm_2 of $REMOVE(x)$, and TA is the set of auxiliary transactions caused by the executions of block ap_1 of $APPEND(x)$ and block rm_1 of $REMOVE(x)$. The occurrence time of each transaction in $TP \cup TA$ is specified by t''. Although each block of the monitor is not executed instantaneously, the execution start time of a block, for example, can be used as the occurrence time of the transaction representing the execution of the block. This is acceptable since each block is executed indivisibly. The system input-list i consists of the input parameters for the $APPEND(x)$ operations, and the system output-list u consists of the values returned by the $REMOVE(x)$ operations.

Let us now define an abstract execution e over level $L = (D, s_0, O)$, where $D = \{\text{buffer}\}$, $\{s_0(\text{buffer}) = \text{empty}\}$, and $O = \{append(x), remove(x)\}$, as follows.

B1) An activation of $APPEND(x)$ with $x = m$ causes an execution of block ap_2. Assume that an abstract operation $append(x)$ with $x = m$ occurs when ap_2 is executed.

B2) An activation of $REMOVE(x)$ causes an execution of block rm_2. Assume that an abstract operation $remove(x)$ occurs when rm_2 is executed.

Then e correctly models e' by data and time abstraction because the following conditions hold.

C1) The condition $s_0 = dabs(s_0')$ is satisfied; i.e., when the monitor is initialized, s_0' defines an empty buffer. Note that $s_0'(COUNT) = 0$.

C2) It is easy to confirm that $APPEND(x)$ correctly implements $append(x)$ under the interpretation given as rule B1. Assume that Append(x) is called with $x = m$, and that $BUFFER[HEAD] @ \cdots @ BUFFER[TAIL] = b$ in state s_1' for which the execution of block ap_2 occurs.[2] Then, in state s_2' produced by the execution of block ap_2, $BUFFER[HEAD] @ \cdots @ BUFFER[TAIL] = b @ m$. Similarly it is easy to confirm that $REMOVE(x)$ correctly implements $remove(x)$ under the interpretation given as rule B2.

C3) Transactions representing the executions of blocks ap_1 and rm_1 have no effects on the abstract buffer state, although they prevent abstract operations from being executed at wrong timings. For example, ap_1 allows an $append(x)$ operation to be applied only to a nonfull buffer.

We have shown that the bounded-buffer monitor given in Fig. 2 is a correct implementation of an abstract buffer of size N under one interpretation. That is, under a different interpretation the bounded-buffer monitor will not work as an abstract buffer of size N. For example, if we assume that each abstract $append(x)$ operation occurs when execution of $APPEND(x)$ is initiated (abstract $remove(x)$ operations are assumed to occur as in our original interpretation), then the effective abstract buffer size becomes greater than N. Note that a process being blocked by waiting for a nonfull condition can effectively buffer one message. Further, messages may be removed from the abstract buffer in a different order from the execution order of the abstract $append(x)$ operations if the wait-signal mechanism uses a scheduling policy other than FCFS.

One interesting feature of our model is that an abstract operation may not be immediately executed when it is issued; its occurrence time is defined by the occurrence times of lower level operations. In our example of the

[2]In general, the system state s_i' seen by a transaction T_i is a (hypothetical) system state in which the effects of all the transactions that logically precede T_i are reflected and the effects of no transactions that logically follow T_i are reflected.

bounded-buffer monitor, an *append*(x) operation, for example, occurs only when the buffer is not full. If the buffer is full when an *append*(x) operation is issued, the operation is delayed by the lower level mechanism; an abstract operation occurs only at an acceptable time.

IV. Multilevel Concurrency Control

In this section we investigate the *multilevel concurrency control* problem of a database system by using data and time abstraction techniques.[3] The major objective of a database system is to maintain data useful to its users. We call such data *logical data*. A typical database system maintains also data that facilitate faster accesses to logical data [1], [31]. We call such data *access path data*. Access path data are often organized into indexing structures like B-trees because of their flexibility and good performance [2]. Logical data and access path data are stored as *physical data*.

At the *logical level* of our multilevel concurrency control of a database system, *user transactions* operate on logical data, and access path data are completely hidden. At the *physical level*, access path data are considered, and operations on access path data are considered correct as long as logical data are correctly accessed. Since operations on access path data need not be serializable in terms of the user transactions that activate them, it is possible to enhance concurrency for access path data by using various techniques [15], [16]. As long as each *logical operation* that manipulates logical data is correctly implemented, we say that *action-level consistency* is maintained [6].

We refer to concurrency control that achieves consistency for user transactions as *long-term concurrency control*, and concurrency control that achieves action-level consistency as *short-term concurrency control*. For expository convenience, we consider only simple locking methods for concurrency control. Locking on logical data by user transactions will be referred to as *long-term locking* (typically, two-phase locking [5]), and that on physical data *short-term locking*.

Unfortunately, a straightforward integration of long-term and short-term concurrency control methods may fail because of the *phantom problem* [5]. Consider a tiny database system that maintains account data for a hypothetical bank. As shown in Fig. 3, each account is represented by a record (*AccountNumber*, *AccountHolder*, *Balance*). Account numbers 00–99 are used for checking accounts, and account numbers 100–199 for savings accounts. The database currently contains records for accounts 10, 30, 110, 120, and 130. Now, execution of transaction T_1 that computes the total asset of each account holder is started. After T_1 has read the records for accounts 10 and 30, transactions T_2 that moves all the money of person B from his savings account to a new checking account is started. T_2 first deletes the record for account 120, and then it inserts the record for account 20. After T_2 is completed,

[3]A preliminary version of this section has appeared in [24].

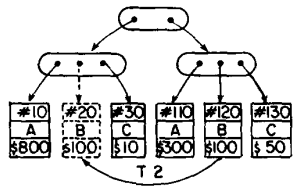


Fig. 3. Phantom problem.

execution of T_1 is continued, and T_1 accesses the records for accounts 110 and 130. If transactions apply long-term locks only to the account records that are accessed by them, T_1 and T_2 do not interfere with each other since they do not access any common account record. In the resultant execution, however, T_1 will report that person B does not have any account. This phenomenon, which is unacceptable, will never happen if T_1 and T_2 are executed one at a time.

The cause for the phantom problem in the above example can be informally explained as follows. Consider again the database given in Fig. 3 and a person, say, person D, for whom no account record is provided. The database implies that person D does not have an account with the bank. However, this information is hidden in the indexing structure. If we examine the indexing structure carefully, we can further localize to the lowest-level access-path objects the places where such information is stored. Note that an account record cannot exist if a pointer to the account record is not contained in any lowest-level access-path object. In order to handle the phantom problem, we will introduce abstract objects *groups* that supposedly store such information. This example shows some subtleties required in deciding what should be treated as logical objects.

A. Logical Level

A *logical database system* contains a set of *data items* and a set of *groups*, and it supports a set of *logical operations* that operate on those data items and groups. Data items and groups are collectively called *logical objects*.

A data item can possess a *data value*. The data value possessed by a data item is called *useless* if it will never be accessed. Otherwise, it is *useful*. The set of data items may be countably infinite, although the number of data items with useful data values must be finite at any given time.

Further, a data item can belong to any finite number (possibly zero) of groups at each given time. When a data item X belongs to a group G, we say that X is a *member* of G. Associations of data items with groups can dynamically change. The set of groups may also be countably infinite, but the number of groups with at least one member must be finite at any given time.

A group can be defined in any conceivable way. A typical case is one where a set of data items possessing a common attribute value form a group. Even a set of data items sharing a page for their physical representations can

form a group. Groups are not mathematical sets since two different groups may contain the same set of members.

The data values possessed by data items can be manipulated by the following logical operations.

Read (X): The current data value of data item X is returned.

Write (X): The data value of data item X is updated to the one supplied; since the data value provided is not relevant for our discussions, we do not show it explicitly by an argument.

Further, data items can be added to and deleted from groups by using the following logical operations.

Insert (X, G): Data item X is made a member of group G.

Remove (X, G): The membership of data item X with group G is resolved.

Although groups can be arbitrarily defined, groups are usually formed so that the members of each group possess some common property. Then the set of data items possessing a certain property can be located by locating the members of the groups associated with that property. The following logical operation is used for locating the members of a group.

MemLocate (G): The names of the data items that are currently members of group G are returned.

Various operations like splitting and merging groups can be implemented by combining *MemLocate* (G), *Remove* (X, G) and *Insert* (X, G) operations.

Assume that data items W and Y are members of group G_1, and that data item Z is the only member of group G_2. At this point, *MemLocate* (G_1) will return the names of W and Y, and *MemLocate* (G_2) the name of Z. Now, consider that *Insert* (X, G_1) is issued. Then X becomes a member of group G_1. If *MemLocate* (G_1) is issued at this point, the names of W, X, and Y will be returned.

We now let *GUpdate* (G) represent either *Insert* (X, G) or *Remove* (X, G) for some X. Note that neither an *Insert* (X, G) operation nor a *Remove* (X, G) operation affects data item X itself. Then, define the relation *conflict* over the set of logical operations as shown in Fig. 4. We leave to the reader the proof that the effects seen by user transactions will not change even if the execution order of any pair of nonconflicting operations are changed. We can regard a *GUpdate* (G) operation as a write operation to logical object G and a *MemLocate* (G) operation as a read operation to logical object G. Although a *GUpdate* (G) operation does not conflict with another *GUpdate* (G) operation unless they manipulate the same member, we do not exploit this property.

An execution of user transactions in which their net effects are as if they were executed one at a time is called *serializable* [5], [27]. Let us call a database system that does not allow dynamic creations or deletions of database entities (in our model, data items and groups) a *static* database system. It is well known that two-phase locking [5] can guarantee a serializable execution for a static database system. Although our model includes an unusual feature (i.e., groups), it still is a static database system,

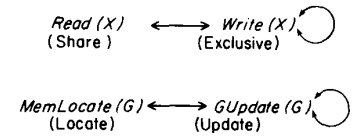


Fig. 4. Conflicting operations and locks used by them.

and hence two-phase locking can still realize a serializable execution, if groups as well as data items are two-phase locked according to relation *conflict*.

Locking consistent with relation *conflict* can be achieved with three lock modes (*Free*, *Share*, and *Exclusive*) provided for data items and with additional three lock modes (*Free*, *Locate*, and *Update*) provided for groups. When a data item or a group is accessed, it must be locked in the mode as indicated in Fig. 4.

We assume that the following logical operations are used for *logical locking*.

MemLock (X, m): Data item X is locked in mode m, which is either *Share* or *Exclusive*.

MemUnlock (X): The lock set on data item X by the user transaction issuing this operation is reset.

GLock (G, m): Group G is locked in mode m, which is either *Locate* or *Update*.

GUnlock (G): The lock set on group G by the user transaction issuing this operation is reset.

If data item X or group G is already locked in a conflicting mode when a *MemLock* (X, m) or *GLock* (G, m) operation is issued, the operation must be blocked, or the user transaction issuing the operation must be aborted.

In a typical database system, data items to be accessed are often designated by specifying their key values or the ranges of their key values. In this paper, a key value is not required to identify a data item uniquely. It is simply a value of an attribute or a set of values of multiple attributes. We now consider a method of defining groups in order to support such value-based accessing. For expository convenience, we consider only one key attribute whose values are totally ordered. The model that satisfies the following rules will be referred to as the *single-key logical database model*.

D1) The *key value* of each data item X is uniquely defined at any time as *key* (X). The key value may vary according to time, and further it may be NUL.[4] The set of all possible key values except for NUL forms a *domain* D. The key values in domain D are totally ordered by $<$. Further, $-\infty < K < +\infty$ for any key value K in D.

D2) Group $G[K]$ is defined for every key value K in D. A data item X such that *key* (X) = K must be a member of group $G[K]$.

Now, we can locate the data items whose current key values are equal to K by locating the members of $G[K]$. Further, we assume that we can locate by an operation *RLocate* (K_i, K_j) the members of the groups whose key values are in the range between K_i and K_j. If there is a countably infinite number of key values in this range, then

[4]An undefined key value must be regarded as NUL.

we must theoretically check the countably infinite number of groups associated with those key values. In Section IV-B we will discuss a method to handle this problem.

B. Physical Level

In this subsection we present a physical level implementation for the single-key logical database model given in Section IV-A. If we want to support multiple access paths, a separate indexing structure must be provided for each key attribute. Note that a data item can be a member of multiple groups.

Since the single-key logical database model allows possibly countable infinite sets of data items and groups, we cannot permanently provide *physical objects* for all of those logical objects. Hence, we dynamically assign physical objects to logical objects and maintain only those physical objects whose values are useful or are different from *default* values.[5] More specifically, a physical object is not provided for a data item with a useless data value or for a group with no members. However, we ensure that the values of logical objects are always uniquely defined unless they are useless, even if their corresponding physical objects do not exist. Since we are assuming that the set of data items that contain useful data values and the set of groups that contain at least one member are both finite at any given time, the number of physical objects thus required is finite.

In order to represent each data item, we use a *target* object. A target object is a physical object of the following format:

record

PLockMode : (Free, Share, Exclusive);

PLockCount : integer;

Refcount : integer;

LLockMode : (Free, Share, Exclusive);

LLockCount : integer;

Value : *ValueType*

end.

The data value of a data item X is stored in the *Value* field of the target object x that represents X. The *LLockMode* field of x shows the current lock mode of X. The *LLockCount* field of x indicates the number of user transactions that currently hold locks on X. When X is locked in *Exclusive* mode, $x.LLockCount$ must be one. The use of other fields will be explained later.

On the other hand, in order to handle a group, a *group descriptor* is provided. A group descriptor is a physical object of the following format:

record

PLockMode : (Free, Share, Exclusive);

PLockCount : integer;

Refcount : integer;

Key : *KeyType*;

GLockMode : (Free, Locate, Update);

GLockCount : integer;

ILockCount : integer;

Members : set of *TargetObjectPtr*

end.

The *Key* field of the group descriptor g for a group $G[K]$ contains the key value K. The identifiers of the target objects that represent the members of $G[K]$ are kept in the *Members* field of g. The *GLockMode* field of g indicates the current lock mode of $G[K]$. The *GLockCount* field of g shows the number of locks being applied to $G[K]$.

According to rule D2, group $G[K]$ is defined for every key value K in the domain D. Then, providing a group descriptor for every group is simply impossible since there can be a countably infinite number of key values. We handle this problem by not providing a group descriptor g such that $g.Refcount = 0$ and $g.Members = \text{NIL}$.[6]

We now discuss an additional locking mechanism for groups. Assume that group descriptors g_i and g_j such that $g_i.Key = K_i$, $g_j.Key = K_j$, and $K_i < K_j$ are provided, and that no group descriptor g such that $K_i < g.Key < K_j$ is provided. When this assumption holds, we say that *interval* $I(K_i, K_j)$ exists. Assume further that a $RLocate(K_i, K_j)$ request is issued. Then, every group $G[K]$ such that $K_i < K < K_j$ must be locked in *Locate* mode.

Let us now consider locking every group $G[K]$ such that $K_i < K < K_j$, or interval $I(K_i, K_j)$, in *Locate* mode, which is the only lock mode possible for an interval. Instead of creating and locking possibly infinite number of group descriptors that should exist between g_i and g_j, we maintain in the *ILockCount* field of g_i the number of locks being applied to these hypothetical group descriptors in interval $I(K_i, K_j)$. When interval $I(K_i, K_j)$ is locked in *Locate* mode, a data item cannot be inserted or removed from any group $G[K]$ such that $K_i < K < K_j$. This locking method will be called *interval locking*.

We now precisely define the default value for a nonexisting group descriptor. Assume that there exist two group descriptors g_i and g_j such that $g_i.Key = K_i$, $g_j.Key = K_j$, and $K_i < K_j$ and that there exists no group descriptor g such that $K_i < g.Key < K_j$. Then, if the group descriptor g' associated with any group $G[K]$ such that $K_i < K < K_j$ would exist,

$g'.Refcount = 0$,

$g'.GLockMode$ = Free if $g_i.ILockCount = 0$,

Locate if $g_i.ILockCount \neq 0$,

[5] A default value may not be fixed. When a default value is not fixed, it must be computable from the values of existing physical objects.

[6] The precise conditions will be given after "interval locks" are introduced.

$g'.GLockCount = g_i.ILockCount$,

$g'.ILockCount = g_i.ILockcount$, and

$g'.Members$ = NIL.

As we stated, we assume that data items to be accessed are designated by specifying their key values. A common method for supporting such an access method is to provide an indexing structure consisting of *access path objects*, which also are physical objects. We assume that access path objects are of the following format:

record

PLockMode : (Free, Share, Exclusive);

PLockCount : integer;

NumberOfSons : 1 .. *MaxFanout*;

Son : array[1..*MaxFanout*] of *ObjectPtr*;

Boundary : array [*1..MaxFanout* − 1] of *KeyType*

end.

Access path objects and group descriptors are organized as a *multiway search tree*.

E1) The search tree is empty or possesses one *root* node.

E2) The root node, if it exists, is either an access path object or a group descriptor.

E3) When an access path object is the root node, it possesses at least two and at most *MaxFanout* descendant nodes. Otherwise, it possesses at least *MinFanout* and at most *MaxFanout* descendant nodes. We assume that *MinFanout* ≤ (*MaxFanout* div 2). A descendant node is either an access path object or a group descriptor.

E4) All leaf nodes of the search tree are group descriptors, and all group descriptors are at the same distance from the root node.

E5) If a group descriptor g can be reached by following *Son*[i] of an access path object p, then $p.Boundary[i-1] < g.Key \leq p.Boundary[i]$ for $p.Boundary[i-1]$ and $p.Boundary[i]$ if they exist.

The condition that *MinFanout* ≤ (*MaxFanout* div 2) guarantees that an access path object with *MaxFanout* descendants can be split into two access path objects of legitimate sizes. This condition, which is slightly different from the one for an ordinary B-tree, is required by the *top-down algorithm* [7] used for tracing the search tree.

When a physical object (an access path object, a group descriptor, or a target object) is accessed, it must be *physically locked* in either *Share* or *Exclusive* mode depending on the mode of the access. The following two operations are used for physical locking.

plock(x, m): The physical object designated by x is physically locked in lock mode m. If it is already locked in a conflicting mode, then the process issuing this operation is blocked. This operation can be implemented as follows.

```
while ((x.PLockMode = Exclusive) or
  (x.PLockMode = Share) and (m = Exclusive))
  do wait;
x.PLockMode := m;
x.PLockCount := x.PLockCount + 1.
```

punlock(x): The physical lock set by the process issuing this operation on physical object x is reset. This operation can be implemented as follows.

```
x.PLockCount := x.PLockCount − 1;
if x.PLockCount = 0 then x.PLockMode := Free.
```

In principle, a physical lock applied to a physical object can be released as soon as the access to the physical object is completed. That is, two-phase locking in terms of user transactions is not necessary. If a physical object is accessed more than once by the same user transaction, the physical object can be locked each time when it is accessed.

We are now ready to discuss implementations of various logical operations. Fig. 5 shows the correspondence between the logical operations and the physical-level procedures that implement them.

Note that logical object identifiers X and G used in Fig. 5 are conceptual (imaginary); only their *surrogates* x and g can actually be used.

In order to locate a target object for a data item X, we must first locate the group descriptor g such that $g.Key = key(X)$. Procedure *glocate*(K) is provided for this purpose.

glocate(K): If the group descriptor g such that $g.Key = K$ already exists, its *Refcount* is incremented by one, and its identifier is returned. If the group descriptor g such that $g.Key = K$ does not exist, it must be created. Let m_I and c_I be the lock mode and the lock count, respectively, of the interval where g falls. That is, if K_i is the largest key value such that $K_i < K$ and for which a group descriptor g_i currently exists, then let m_I = Free when $g_i.ILockCount = 0$ or m_I = Locate when $g_i.ILoclcount \neq 0$, and let $c_I = g_i.ILockCount$. Now, g can be created with the following field values.

PLockMode = Free,

PLockCount = 0,

Refcount = 1,

Key = K,

GLockMode = m_I,

GLockCount = c_I,

ILockCount = c_I,

Members = NIL.

After g is inserted into the search tree, the identifier of g is returned.

var X	<--->	x :- *new(TargetObjectType)*
		g :- *glocate(K)*;
{ X } :- *MemLocate(G[K])*	<--->	{ x } :- *memlocate(g)*
(no counterpart)	<--->	*grelease(g)*
(no counterpart)	<--->	*memrelease(x)*
Read(X)	<--->	*read(x)*
Write(X)	<--->	*write(x)*
Insert(X, G)	<--->	*insert(x, g)*
Remove(X, G)	<--->	*remove(x, g)*
MemLock(X, m)	<--->	*memlock(x, m)*
MemUnlock(X)	<--->	*memunlock(x)*
GLock(G, m)	<--->	*glock(g, m)*
GUnlock(G)	<--->	*gunlock(g)*

Fig. 5. Logical operations and their associated physical-level procedures.

Function *glocate* (K) must scan the search tree starting from the root node until a group descriptor is reached. The implementation of *glocate* (K) shown in Fig. 6 uses the top-down algorithm given in [7].[7] Only exclusive locks are used for short-term locking. If an access path object with too many ($=MaxFanout$) descendants is encountered, the access path object is split. If an access path object with too few ($= MinFanout$) descendants is encountered, the access path object is merged with its neighbor or some descendant pointers in its neighbor are moved to the access path object. (Although the root node must be treated differently, we do not discuss the details.) Further, physical locks on access path objects are seized and released according to the *tree protocol* of [29].

When group descriptor g such that $g.Key = K$ is reached, the set of target objects that represent the data items belonging to $G[K]$ can be located.

memlocate (g): The target object identifiers in $g.Members$ are returned. When the identifier of a target object x is returned, $x.Refcount$ is incremented by one.

If a target object x does not exist for a data item X, then x can be created as "x :- *new(TargetObjectType)*."[8]

Once the target object x for a data item X is known, the data value of X can be accessed.

read (x): $X.Value$ is returned.

write (x): $x.Value$ is updated to the data value provided.

Assume that target object x represents a data item X, and group descriptor g represents a group G. Then, *Insert* (X, G) and *Remove* (X, G) operations can be implemented as follows.

insert (x, g):

$$g.Members := g.Members \cup \{x\};$$
$$x.Refcount := x.Refcount + 1.$$

[7]Obviously, any algorithm that guarantees consistency for search tree accesses can be used. See [15], [16] for various algorithms that can be used for this purpose.

[8]":-" is the Simula notation for the assignment operator for a pointer value.

```
function glocate(K: KeyType): GroupDescriptorType;

begin
 LastNodePtr :- nil;
 plock(LastNodePtr↑);   (* lock RootNodePtr *)
 NodePtr :- RootNodePtr; (* start with the root node *)
 plock(NodePtr↑);       (* lock the root node *)
 while NodePtr↑ is an access path object do
  begin
   case
    NodePtr↑ is too big:
     begin
      split NodePtr↑;
      punlock(NodePtr↑);
      adjust NodePtr; (* put NodePtr on the right path *)
      plock(NodePtr↑);
     end;
    NodePtr↑ is too small:
     begin
      plock(neighbor of NodePtr↑);
      merge NodePtr↑ with its neighbor or move
      some son pointers of the neighbor to NodePtr↑;
      punlock(neighbor of NodePtr↑);
      punlock(NodePtr↑);
      adjust NodePtr; (* put NodePtr on the right path *)
      plock(NodePtr↑);
     end;
   end;
   punlock(LastNodePtr↑);
   LastNodePtr :- NodePtr;
   NodePtr :- NodePtr↑.Son[Rank] where
     Boundary[Rank-1] < K < Boundary[Rank] for NodePtr↑;
   plock(NodePtr↑);
  end;

 (* group descriptor is reached *)
 if NodePtr↑.Key = K then
   begin   (* group descriptor for K already exists *)
     punlock(LastNodePtr↑);
     NodePtr↑.RefCount := NodePtr↑.RefCount + 1;
     glocate :- NodePtr;
   end
 else
   begin   (* group descriptor for K does not exist *)
     create a group descriptor NewLeaf↑
       with Key = K, RefCount = 1, ... ;
     insert the pointer to NewLeaf↑ into LastNodePtr↑;
     punlock(LastNodePtr↑);
     glocate :- NewLeaf;
   end;
 punlock(NodePtr↑);
end
```

Fig. 6. Function glocate(K).

remove (x, g):

$$g.Members := g.Members - \{x\};$$
$$x.Refcount := x.Refcount - 1.$$

A target object and a group descriptor must be released after their use. A target object can be released as follows.

memrelease(x): $x.Refcount$ is decremented by one. If the resultant $x.Refcount$ is zero, then x is deleted.

If $x.Refcount = 0$, then neither x belongs to any group, nor its identifier is held by any process. Therefore, x will never be accessed, and hence it can be deleted. Note that when $x.Refcount = 0$, $x.LLockCount$ must be zero. This requirement is natural since a lock on x cannot be released if x has been released.

Our implementation does not allow target objects to be deleted explicitly. However, a request for a target object deletion can be supported as follows. Assume that NUL is the data value to be returned when a *read* (x) operation is applied to a nonexisting target object x. Then, x is effectively deleted if NUL is assigned to $x.Value$. A target object containing such NUL value can be regarded as a "tombstone" [20].

A group descriptor g seized by a *glocate*(K) operation can be released by a *grelease*(g) operation.

grelease(g): $g.Refcount$ is decremented by one. Let g_i be the group descriptor that immediately precedes g (i.e., $g_i.Key < g.Key$, and $g_i.Key < g'.Key < g.Key$ for no existing group descriptor g'). Now, if the following condition holds, g can be deleted:

$g.Refcount = 0$,

$g.GLockMode$ = Free if $g_i.ILockCount = 0$,

Locate if $g_i.ILockCount \neq 0$,

$g.GLockCount = g_i.ILockCount$,

$g.ILockCount = g_i.ILockCount$, and

$g.Members$ = NIL.

In order for the top-down algorithm to work correctly, each *grelease*(g) must be preceded by a *glocate'*(K) operation such that $K = g.Key$ if the deletion of g is expected. This operation must work like a *glocate*(K) operation except that it does not increment $g.Refcount$. A *glocate'*(K) operation prevents access path objects from possessing too few descendants, as well as it locates the immediate ancestor node of g from which the pointer to g is deleted.

Logical locking must be performed by regarding that each *Read*(X) or *Write*(X) operation occurs when its corresponding *read*(x) or *write*(x) operation occurs. The reason why this rule works correctly is discussed in Section IV-C. Let x be the target object representing a data item X. Then, logical operations *MemLock*(X, m) and *MemUnlock*(X) can be performed by physical operations *memlock*(x, m) and *memunlock*(x), respectively.

memlock(x, m):

```
while ((x.LLockMode = Exclusive) or
  (x.LLockMode = Share) and (m = Exclusive))
  do wait;
x.LLockMode := m;
x.LLockCount := x.LLockCount + 1.
```

memunlock(x):

```
x.LLockCount := x.LLockCount - 1;
if x.LLockCount = 0 then x.LLockMode := Free.
```

Group locking can be performed similarly. Let g be the group descriptor representing a group G. Then, logical operations *GLock*(G, m) and *GUnlock*(G) can be performed by physical operations *glock*(g, m) and *gunlock*(g), respectively.

glock(g, m):

```
while ((g.GLockMode = Update) or
  (g.GLockMode = Locate) and (m = Update))
  do wait;
g.GLockMode := m;
g.GLockCount := g.GLockCount + 1.
```

gunlock(g):

```
g.GlockCount := g.GLockCount - 1;
if g.GLockCount = 0 then g.GLockMode := Free.
```

Assume that for a pair of group descriptors g_i and g_j, $g_i.Key = K_i$, $g_j.Key = K_j$, and $K_i < K_j$, and further that there exists no group descriptor g such that $K_i < g.Key < K_j$. Then, interval $I(K_i, K_j)$ can be locked and unlocked as follows.

ilock(g_i):

```
g_i.ILockCount := g_i.ILockCount + 1.
```

iunlock(g_i):

```
g_i.ILockCount := g_i.ILockCount - 1.
```

Assume that a single data item X such that $key(X) = K$ must be updated. An example of an execution at the logical level and its counterpart at the physical level is given in Fig. 7. Note that logical locks are applied according to the two-phase locking rule.

In Fig. 8, the periods of the logical locks and the physical locks applied during an execution of a user transaction whose logical representation is *Read*(X); *Read*(Y); *Write*(Z) is shown, where $key(X) = K_1$, $key(Y) = K_2$, $key(Z) = K_3$. Note that physical locks are applied for far shorter periods than logical locks.

In Fig. 9, the database state histories created by three different ways of interleaving of logical operations are shown. The logical write operations of T_1 and T_2 do not conflict with each other, and hence they may interleave in any way. Note that *Write*(B), for example, must be executed as follows:

```
g_B :- glocate('B');
b :- New(TargetObjectType);
write(b);
insert(b, g_B);
grelease(g_B).
```

Target objects are not shown in Fig. 9.

The tree structure of Fig. 9(h), which results if T_1 and T_2 are executed in the order of *Write*(B), *Write*(D), *Write*(I), and *Write*(G), cannot occur if T_1 and T_2 are executed one at a time; either Fig. 9(e) or Fig. 9(l) will result. Note that the tree structure in Fig. 9(h) represents the same logical database state as the tree structure in Fig. 9(e) or Fig. 9(l), and that it must be considered correct.

C. Correctness

In this subsection, we show that the implementation of a multilevel concurrency control scheme given in the preceding subsection is correct. First, it is proved that logical objects and logical operations are correctly implemented by physical objects and physical operations. Then, we show that logical operations will be correctly scheduled if consistent locking is used for logical operations.

We first show that logical objects and logical operations applied to them are correctly implemented. For this pur-

	g := *glocate*(*K*);
GLock(*G*[*K*], Locate);	*glock*(*g*, Locate);
	{ *x* } := *memlocate*(*g*);
MemLock(*X*, Exclusive);	*memlock*(*x*, Exclusive);
GUnlock(*G*[*K*]);	*gunlock*(*g*);
	grelease(*g*);
Read(*X*);	*read*(*x*);
.	.
.	.
Write(*X*);	*write*(*x*);
MemUnlock(*X*);	*memunlock*(*x*);
	memrelease(*x*);

Fig. 7. A logical execution and its physical counterpart.

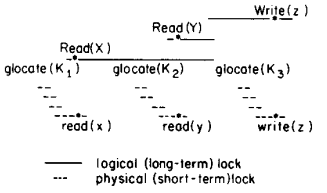


Fig. 8. Long-term and short-term locking.

pose, we consider the data abstraction function that associates logical objects with physical objects as follows.

F1: Each target object x represents a *separate* data item X.[9] If a target object x exists for a data item X, then the data value of X is defined by $x.Value$, and the lock status of X is defined by $x.LLockMode$ and $x.LLockCount$. If such x does not exist, then the data value of X is undefined, and no locks are applied on X.

F2: If a group descriptor g such that $g.Key = K$ exists for a key value K, then $g.Members$ contains the identifiers of the target objects that represent the members of group $G[K]$, and the lock status of $G[K]$ is shown by $g.GLockMode$ and $g.GLockCount$. If such g does not exist for a key value K, then $G[K]$ is empty. In this case, the lock status of $G[K]$ can be known from $g_i.ILockCount$, where g_i is the group descriptor that would immediately precede g if there were g.

An implementation of logical operations is correct if the following conditions are satisfied.

G1: A logical write operation *Write*(X) correctly updates the data value of X as defined by F1, and a logical read operation *Read*(X) returns the current data value of X as defined by F1.

G2: Each group $G[K]$ as defined by F2 is properly accessed by *Insert*(X, $G[K]$), *Remove*(X, $G[K]$) and *MemLocate*($G[K]$) operations as discussed in Section IV-A.

Now, the following lemma is trivially true.

Lemma 1: *Read*(X) and *Write*(X) are properly implemented by *read*(x) and *write*(x), respectively where x is the target object for data item X.

Note that a data item with an undefined data value will never be accessed, since its target object cannot be reached.

[9]If a target object x_2 is not a continuation of a target object x_1, they represent different data items even if they are an identical data item from the user's viewpoint.

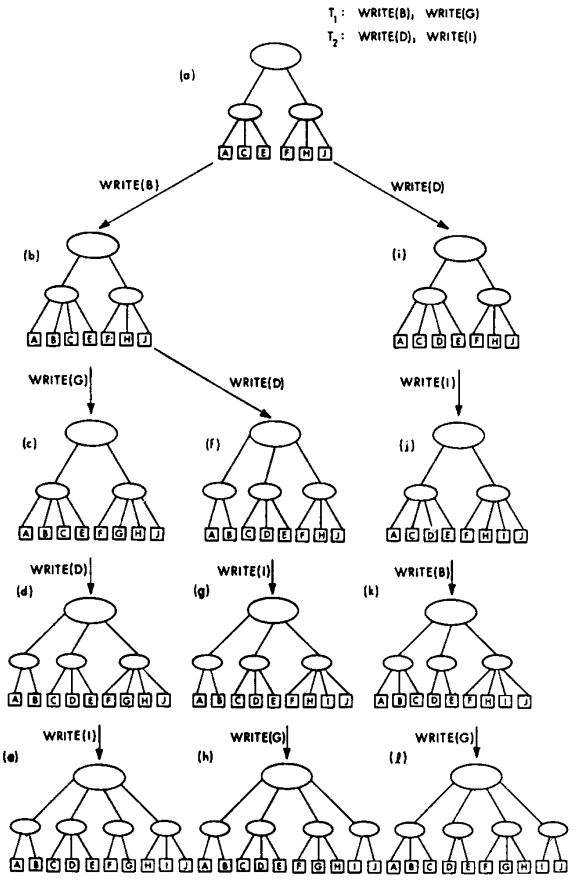


Fig. 9. Physically nonserializable execution.

The following lemma is also trivial.

Lemma 2: *Insert*(X, G), *Remove*(X, G) and *MemLocate*(G) are correctly implemented by *insert*(x, g), *remove*(x, g), and *memlocate*(g), respectively, where x and g are the target object and the group descriptor that respectively represent X and G.

We say that *glocate*(K) and *grelease*(g) are correctly implemented, if they satisfy the following requirements.

H1: At any time at most one group descriptor exists in the system for each group $G[K]$.

H2: For each pair of *glocate*(K) and *grelease*(g) issued by a user transaction, group descriptor g returned by *glocate*(K) is the correct group descriptor for $G[K]$ until the corresponding *grelease*(g) occurs.

H3: The state of $G[K]$ is continuous when its group descriptor is created or deleted.

Although we do not show the detailed implementations of procedure *grelease*(g), we assume that *glocate*(K) and *grelease*(g) satisfy the specifications given in Section IV-B. Then, we have the following lemma, which concerns correctness of sequential programs.

Lemma 3: Requirements H1, H2, and H3 are satisfied if *glocate*(K) [and *glocate'*(K)] and *grelease*(g) are executed one at a time.

Proof: Requirements H1 and H2 immediately follow from the specifications for *glocate*(K) and *grelease*(g). Requirements H3 follows from the fact that the value of a group descriptor created or deleted is identical to the default value defined for the group descriptor. □

Further, requirements H1, H2, and H3 are still satisfied even when *glocate*(K) and *grelease*(g) are executed concurrently.

Lemma 4: Even if procedures *glocate*(K) and *grelease*(g) are executed concurrently, they produce the same effects as when they are executed one at a time.

Proof: The implementation of *glocate*(K) and *grelease*(g) follows the tree protocol of [29], and hence their execution is *serializable in terms of these operations.* □

If logical operations are correctly implemented and if the execution of logical operations is *serializable in terms of user transactions*, then we can consider that the resultant system operation is correct. This condition is satisfied if a consistent locking scheme is employed for logical objects. The point here is that accesses to physical objects (in particular, to access path objects) need not be serializable in terms of user transactions.

In order to define the execution of logical operations, their occurrence times can be specified as follows.

Definition (Execution Times of Logical Operations): The occurrence time of a logical operation *Read*(X) or *Write*(X) is defined to be the time when the target object x for logical object X is accessed by the physical operation *read*(x) or *write*(x) that implements *Read*(X) or *Write*(X). Similarly, the occurrence time of a logical operation *MemLocate*(G), *Insert*(X, G) or *Remove*(X, G) is defined to be the time when the group descriptor g that represents G is accessed by the physical operation *memlocate*(g), *insert*(x, g), or *remove*(x, g) that implements the logical operation.

Once occurrence times of logical operations are defined as above, the following lemma is immediate from the implementations of logical operations.

Lemma 5: The values of logical objects as defined by F1 and F2 are accessed by logical operations exactly at the points when those logical operations are supposed to occur according to the above definition.

We now can conclude that logical objects can be regarded as real objects as far as logical operations are concerned. Hence, if logical operations are performed under a consistent (logical) locking scheme, the resultant execution will be serializable at the logical level.

V. Conclusion

We have presented a framework for the design, implementation, and analysis of a multilevel concurrent software system. Time abstraction used in combination with data abstraction plays a key role in our methodology. Time abstraction allows us to specify explicitly the execution times of abstract operations. Abstract operations implemented in this way can be readily used, and they can be synchronized according to their hypothetical execution times.

Although we have not shown any examples, concurrent background processing like garbage collection and data-structure reorganization can be discussed within the framework presented in Section II. Time abstraction has been implicit in such concepts as atomicity, timestamp-based concurrency control, and multiversion concurrently control. We consider that its usefulness can be further exploited by its full recognition.

Acknowledgment

The authors wish to thank the anonymous referees for their constructive comments.

References

[1] M. M. Astrahan, *et al.*, "System R: Relational approach to database management," *ACM Trans. Database Syst.*, vol. 1, pp. 97–137, June 1976.

[2] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173–189, 1972.

[3] C. Beeri, P. A. Bernstein, and N. Goodman, "A model for concurrency in nested transactions systems," Dep. Comput. Sci., Hebrew Univ., Rep. TR CS-86-1, 1986.

[4] P. Bernstein, D. Shipman, and W. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Trans. Software Eng.*, SE-5, pp. 203–216, May 1979.

[5] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, pp. 624–633, Nov. 1976.

[6] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the System R database manager," *Comput. Surveys*, vol. 13, pp. 223–242, June 1981.

[7] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Symp. Foundations Comp. Sci.*, 1978, pp. 8–21.

[8] J. Guttag, "Abstract data types and the development of data structures," *Commun. ACM*, vol. 20, pp. 396–404, June 1977.

[9] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Commun. ACM*, vol. 21, pp. 1048–1064, Dec. 1978.

[10] J. Guttag, "Notes on type abstraction (version 2)," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 13–23, Jan. 1980.

[11] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Inform.*, vol. 1, pp. 271–281, 1972.

[12] ——, "Monitors: An operating system structuring concept," *Commun ACM*, vol. 17, pp. 549–557, Oct. 1974.

[13] J. H. Howard, "Proving monitors," *Commun. ACM*, vol. 19, pp. 273–279, May 1976.

[14] Y. S. Kwong, "On reduction of asynchronous systems," *Theoretical Comput. Sci.*, vol. 5, pp. 25–50, 1977.

[15] Y. S. Kwong and D. Wood, "A new method for concurrency in B-Trees," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 211–222, May 1982.

[16] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Trans. Database Syst.*, vol. 6, pp. 650–670, Dec. 1981.

[17] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, pp. 717–721, Dec. 1975.

[18] B. H. Liskov and S. N. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7–19, Mar. 1975.

[19] B. Liskov and A. Snyder, "Abstraction mechanisms in CLU," *Commun. ACM*, vol. 20, pp. 564–576, Aug. 1977.

[20] D. B. Lomet, "Scheme for invalidating references to freed storage," *IBM J. Res. Develop.*, vol. 19, pp. 26–35, Jan. 1975.

[21] ——, "Process structuring synchronization, and recovery using atomic actions," *SIGPLAN Notices*, vol. 12, pp. 128–137, 1977.

[22] J. McCarthy, "Toward a mathematical science of computation," in *Proc. IFIP*, 1962, pp. 21–28.

[23] T. Minoura, "Time abstraction," unpublished draft, Univ. Southern Calif., 1981.

[24] ——, "Multi-level concurrency control of a database system," in

Proc. 4th Symp. Reliability in Distributed Software and Database Systems, 1984, pp. 156–168.

[25] J. E. Moss, N. D. Griffeth, and M. H. Graham, "Abstraction in recovery management," in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, 1986, pp. 72–83.

[26] S. S. Owicki, "Specifications and proofs for abstract data types in concurrent programs," Comput. Syst. Lab., Standford Univ., Tech. Rep. 133, 1977.

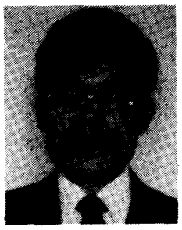
[27] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[28] D. P. Reed, "Implementing atomic actions on decentralized data," *ACM Trans. Comput. Syst.*, vol. 1, pp. 3–23, Feb. 1983.

[29] A. Silberschatz and Z. Kedem, "Consistency in hierarchical database systems," *J. ACM*, vol. 27, pp. 72–80, Jan. 1980.

[30] W. A. Wulf, R. L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 253–265, Dec. 1976.

[31] S. B. Yao, "An attribute based model for database access cost analysis," *ACM Trans. Database Syst.*, vol. 2, pp. 45–67, Mar. 1977.



Toshimi Minoura (S'76–M'79–S'79–M'80) received the B.S. and M.S. degrees from Tokyo University, Tokyo, Japan, in 1968 and 1970, respectively, and the Ph.D. degree from Stanford University, Stanford, CA, in 1980, all in electrical engineering.

He is currently an Associate Professor of Computer Science at Oregon State University, Corvallis. He has conducted research on deadlock problems, true-copy token schemes, multiversion and multilevel concurrency control, and reliable storage subsystems. He recently started to work on 4GL approach for CAD.

S. Sitharama Iyengar received the Ph.D. degree in engineering in 1974.

He is currently a Professor of Computer Science and Supervisor of robotic research and parallel algorithms at Louisiana State University, Baton Rouge. He has authored (coauthored) more than 75 research papers in parallel algorithms, data structures, navigation of intelligent mobile robots, etc. He is currently studying the application of neural network techniques for path planning and learning in mobile robots. His papers have appeared in the following journals: IEEE TSE, IEEE PAMI, IEEE SMC, IEEE JOURNAL OF ROBOTICS AND AUTOMATION, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *CACM*, *JCIS*, *Computer Networks*, *Journal of Robotic Systems*, *BIT*, *Theoretical Computer Science*, and several other international journals and IEEE proceedings. He is an ACM National Lecturer for 1986–1988. His research has been funded by NASA, DOE, NAVY, Jet Propulsion Lab., and Caltech.

Dr. Iyengar has been on program committees for several major conferences in the USA and in Europe. He was a Co-Guest Editor for a special issue of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He is also a guest editor for a special issue on autonomous intelligent machines in IEEE *Computer* magazine.