INTERNATIONAL JOURNAL OF NATURAL AND APPLIED SCIENCES (IJNAS), VOL. 6, NOS.1& 2 (2011); P. 63 – 71, 1TABLE,
.

Issues on concurrency controls in transactional database system: The automatic teller machine (ATM) problem.

F. U. Ogban[1*] and P. Asagba[2]

ABSTRACT

Concurrency control issues like the "isle-of-March" has come but not gone. It is not surprising that several models designed to handle concurrency control in databases; there still exist some overheads in transactional database controls. One of such problems is the incompletion of the execution of necessary codes within the critical region of a lock transaction implementation thus generating data integrity fault (–such as dirty read, non-repeated read and phantom read) and inconsistent data (- such as non-submitted data, late submitted data etc). In this work, the locks paradigms were reappraised to suit situations of usage and recommend particular areas of application of the different locking strategies so designed. New construct of control issues were recreated vis-à-vis the application situation found on ground like the Automatic Teller Machine problem.

INTRODUCTION

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that *may* be executed in parallel (Taubenfeld, 2006). Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network (Ben-Ari, 2006). Above understanding if considered strictly, would limit our avenues for implementing concurrent computing in some instance. It must be noted that the main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes. In such design, especially in transactional computing, where database contents are read and written, our target is to avoid interferences of a process (with its associated data) and other processes (with its own associated data). A number of different methods can be used to implement concurrent programs, such as implementing each computational process as an operating system process. (Brinch, 1993, Hoare, 1974) or implementing the computational processes as a set of threads within a single operating system process (Taubenfeld; 2006).

The concept of nested transactions offers more decomposable execution units and finer-grained control over concurrency and recovery than "flat" transactions. Furthermore, it supports the decomposition of a "unit of work" into subtasks and their appropriate distribution in a computer system as a prerequisite of intra-transaction parallelism. However, to exploit its full potential, suitable granules of concurrency control as well as access modes for shared data are necessary (Ben-Ari, 2006). In this paper, we investigate various issues of concurrency control for nested transactions.

Communication in concurrent computing

In some concurrent computing systems, communication between the concurrent components is hidden from the programmer (e.g., by using futures), while in others it must be handled explicitly. Explicit communication can be divided into two classes as fellows

a).Shared memory communication: Concurrent components communicate by altering the contents of shared memory locations (exemplified by Java and C#). This style of concurrent programming usually requires the application of some form of locking (e.g., mutexes (means mutual exclusion), semaphores, or monitors) to coordinate between threads.

b). Message passing communication: Concurrent components communicate by exchanging messages (exemplified by Erlang and occam). The exchange of messages may be carried out asynchronously (sometimes referred to as "send and pray", although it is standard practice to resend messages that are not

*Corresponding author**.** Email:
[1]Department of Mathematics/Statistics & Computer Science, University of Calabar, Calabar, Nigeria
[2]Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria

acknowledged as received), or may use a rendezvous style in which the sender blocks until the message is received. Message-passing concurrently tends to be far easier to reason about than shared-memory concurrency, and it is typically considered more robust, although slower form of concurrent programming (Philip et al, 1987).

A wide variety of mathematical theories for understanding and analyzing message-passing systems are available, including the Actor model, and various process calculi. Message passing can be efficiently implemented on symmetric multiprocessors using shared coherent memory (Wilson, 1994).

Coordinating access to resources

One of the major issues in concurrent computing is preventing concurrent processes from interfering with each other. For example, consider the following algorithm for making withdrawals from a checking account represented by the shared resource balance a typical situation of what happens in an Automated Teller Machine (ATM) system:

```
1  bool withdraw(int withdrawal)
2  {
3     if( balance >= withdrawal )
4     {
5        balance -= withdrawal;
6        return true;
7     }
8     return false;
9  }
```

Suppose balance=1500, and two concurrent processes make the calls ***withdraw(1300)*** and ***withdraw(350)***. If line 3 in both operations executes before line 5 both operations will find that '***balance*** > ***withdrawal***' evaluates to true, and execution will proceed to subtracting the withdrawal amount. However, since both processes perform their withdrawals, the total amount withdrawn will end up being more than the original balance. These sorts of problems with shared resources require the use of concurrency control, or non-blocking algorithms. Because concurrent systems rely on the use of shared resources (including communications mediums), concurrent computing in general requires the use of some form of arbiter somewhere in the implementation to mediate access to these resources (Buhr et al., 1995).

Unfortunately, while many solutions exist to the problem of a conflict over one resource, many of those "solutions" have their own concurrency problems such as deadlock when more than one resource is involved.

METHODOGOLY

- We would propose a collection of concurrent computing models suitable for the ATM Machine and targeted at avoiding the dirty read, phantom read and non-repeated read problems which causes double withdrawal entry, submitted non-withdrawn and debited non withdrawn cases in our banks.
- A model for nested transactions is proposed allowing for effective exploitation of intra-transaction parallelism. Starting with a set of basic locking rules, we introduce the concept of "downward inheritance of locks" to make data manipulated by a parent available to its children. To support supervised and restricted access, this concept is refined to "controlled downward inheritance" .
- We would present pseudo codes that would provoke more appropriate but classical program structures to favor critical region modularity.

Concurrent programming languages

Concurrent programming languages are programming languages that use language constructs for concurrency. These constructs may involve multi-threading, support for distributed computing, message passing, shared resources (including shared memory) or futures (known also as *promises*).

Today, the most commonly used programming languages that have specific constructs for concurrency are Java and C#. These two languages fundamentally use a shared-memory concurrency model, with locking provided by monitors (however message-passing models can and have been implemented on top of the underlying shared-memory model). Of the languages that use a message-passing concurrency model, Erlang is probably the most widely used in industry at present.

Many concurrent programming languages have been developed more as research languages (e.g. Pict) rather than as languages for production use. However, languages such as Erlang, Limbo, and occam have seen industrial use at various times in the last 20 years. Languages in which concurrency plays an important role include:

- ActorScript This is a theoretical and purely an actor-based language, defined in terms of itself – recursive.
- Afnix – Concurrent access to data is protected automatically (previously called *Aleph*, but unrelated to *Alef*)
- Alef – Concurrent language with threads and message passing, used for systems programming in early versions of Plan 9 from Bell Labs
- Eiffel – Through its SCOOP mechanism based on the concepts of Design by Contract

- Erlang – Uses asynchronous message passing with nothing shared
- Limbo – Relative of Alef, used for systems programming in Inferno (operating system)
- MultiLisp – Scheme variant extended to support parallelism
- Modula-3 – Modern language in Algol family with extensive support for threads, mutexes, condition variables.
- occam – Influenced heavily by Communicating Sequential Processes (CSP).
    - occam-π – a modern variant of occam, which incorporates ideas from Milner's π-calculus

Models of concurrency

There are several models of concurrent computing, which can be used to understand and analyze concurrent systems. These models include:

- The Actor model
- Petri nets
- Process calculi such as
    - Ambient calculus
    - Calculus of communicating systems (CCS)
    - Communicating sequential processes (CSP)
    - π-calculus
- Ptolemy Project
- Race condition
- Critical section
- Transaction processing
- Software transactional memory
- Flow-based programming

Concurrency control

In computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

Concurrency control in databases

Concurrency control in database management systems (DBMS) ensures that database transactions are performed concurrently without violating the data integrity of a database. Executed transactions should follow the ACID rules, as described below. The DBMS must guarantee that only serializable (unless Serializability is intentionally relaxed), recoverable schedules are generated. It also guarantees that no effect of committed transactions is lost, and no effect of aborted (rolled back) transactions remains in the related database.

Concurrency control mechanism

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the synchronization for a transaction until its end without blocking (read, write) operations, and then abort transactions that violate desired synchronization rules.
- **Pessimistic** - Block operations of transaction that would cause violation of synchronization rules.

Many methods for concurrency control exist. Major methods, which have each many variants, include:

- Two phase locking
- Conflict (serializability, precedence) graph checking
- Timestamp ordering
- Commitment ordering
- Multiversion concurrency control
- Index concurrency control (for synchronizing indexes)

Almost all implemented concurrency control mechanisms, typically quite efficient, guarantee schedules that are conflict serializable, unless relaxed forms of serializability are allowed (depending on application) for better performance.

Collisions

To understand how to implement concurrency control within your system you must start by understanding the basics of collisions – you can either avoid them or detect and then resolve them. The next step is to understand transactions, which are collections of actions that potentially modify two or more entities. One important message of this work is that on modern software development projects, concurrency control and transactions are not simply the domain of databases, instead they are issues that are potentially pertinent to all of your architectural tiers. (Acken et al., 1998)

In Implementing Referential Integrity and Shared Business Logic we have to understand the referential integrity challenges that result from there being an object schema that is mapped to a data schema, this was earlier referred to as cross-schema referential integrity problems (Hoare; 1974). With respect to collisions things are a little simpler; we only need to worry about the issues with ensuring the consistency of entities within the system of record (Asonovic et al; 2006). The system of record is the location where the official version of an entity is located. This is often data stored within a relational database although other representations, such as an XML structure or an object, are also viable.

A collision is said to occur when two activities, which may or may not be full-fledged transactions, attempt to change entities within a system of record. There are three fundamental ways (Culler et al., 1999) that two activities can interfere with one another; namely Dirty, non-repeatable and phantom read.

i. **Dirty read**. Activity 1 (A1) reads an entity from the system of record and then updates the system of record but does not

commit the change (for example, the change hasn't been finalized). Activity 2 (A2) reads the entity, unknowingly making a copy of the uncommitted version. A1 rolls back (aborts) the changes, restoring the entity to the original state that A1 found it in. A2 now has a version of the entity that was never committed and therefore is not considered to have actually existed.

ii. **Non-repeatable read**. A1 reads an entity from the system of record, making a copy of it. A2 deletes the entity from the system of record. A1 now has a copy of an entity that does not officially exist.

iii. **Phantom read**. A1 retrieves a collection of entities from the system of record, making copies of them, based on some sort of search criteria such as "all customers with first name Bassey."A2 then creates new entities, which would have met the search criteria (for example, inserts "Bassey Kanu" into the database), saving them to the system of record. If A1 reapplies the search criteria it gets a different result set.

In our case with the ATM machine, the problem is not that of reading the memory object, executing the deduction therein and committing same but mechanically not dispensing the cash thereafter.

Locking strategies

So what can you do? First, you can take a pessimistic locking approach that avoids collisions but reduces system performance. Second, you can use an optimistic locking strategy that enables you to detect collisions so you can resolve them. Third, you can take an overly optimistic locking strategy that ignores the issue completely. The second is most preferred since we can by detection, resolve the collisions. Because the cost of systems performance arising from the first and that of totally ignoring the issue completely as proposed by the third, would be more weighty than the cost of resolving the collision.

Pessimistic locking

Pessimistic locking is an approach where an entity is locked in the database for the entire time that it is in application memory (often in the form of an object). A lock either limits or prevents other users from working with the entity in the database. A write lock indicates that the holder of the lock intends to update the entity and disallows anyone from reading, updating, or deleting the entity. A read lock indicates that the holder of the lock does not want the entity to change while the hold the lock, allowing others to read the entity but not update or delete it. The scope of a lock might be the entire database, a table, a collection of rows, or a single row. These types of locks are called database locks, table locks, page locks, and row locks respectively.

Optimistic locking

With multi-user systems it is quite common to be in a situation where collisions are infrequent. Although the two of us are working with *Customer* objects, you're working with the Bassey Okon's object while I work with the John Etim's object and therefore we won't collide. When this is the case optimistic locking becomes a viable concurrency control strategy. The idea is that you accept the fact that collisions occur infrequently, and instead of trying to prevent them you simply choose to detect them and then resolve the collision when it does occur.

There are two basic strategies for determining if a collision has occurred:

1. **Mark the source with a unique identifier**. The source data row is marked with a unique value each time it is updated. At the point of update, the mark is checked, and if there is a different value than what you originally read in, then you know that there has been an update to the source. There are different types of concurrency marks:
    - Datetime stamps (the database server should assign this value because you can't count on the time clocks of all machines to be in sync).
    - Incremental counters.
    - User IDs (this only works if everyone has a unique ID and you're logged into only one machine and the applications ensure that only one copy of an object exists in memory).
    - Values generated by a globally unique surrogate key generator.
2. **Retain a copy of the original**. The source data is retrieved at the point of updating and compared with the values that were originally retrieved. If the values have changed, then a collision has occurred. This strategy may be your only option if you are unable to add sufficient columns to your database schema to maintain the concurrency marks.

Overly optimistic locking

With the strategy you neither try to avoid nor detect collisions, assuming that they will never occur. This strategy is appropriate for single user systems, systems where the system of record is guaranteed to be accessed by only one user or system process at a time, or read-only tables. These situations do occur. It is important to recognize that this strategy is completely inappropriate for multi-user systems.

Collision resolution strategies

You have five basic strategies that you can apply to resolve collisions:

1. **Give up**.
2. **Display the problem and let the user decide**.
3. **Merge the changes**.

4. **Log the problem so someone can decide later**.
5. **Ignore the collision and overwrite**.

It is important to recognize that the granularity of a collision counts. Assume that both of us are working with a copy of the same *Customer* entity. If you update a customer's name and I update their shopping preferences, then we can still recover from this collision. In effect the collision occurred at the entity level, we updated the same customer, but not at the attribute level. It is very common to detect potential collisions at the entity level then get smart about resolving them at the attribute level.

Choosing a locking strategy

For simplicity's sake, many project teams will choose a single locking strategy and apply it for all tables. This works well when all, or at least most, tables in your application have the same access characteristics. However, for more complex applications you will likely need to implement several locking strategies based on the access characteristics of individual tables. One approach, suggested by Willem Bogaerts, is to categorize each table by type to provide guidance as to a locking strategy for it. Strategies for doing so are described in Table 1.

Table 1. Locking strategies by table type.

Table Type	Examples	Suggested Locking Strategy
Live-High Volume	• Account	• Optimistic (first choice) • Pessimistic(second choice)
Live-Low Volume	• Customer • Insurance Policy	• Pessimistic (first choice) • Optimistic (second choice)
Log (typically append only)	• AccessLog • AccountHistory • TransactionRecord	• Overly Optimistic
Lookup/Reference (typically read only)	• State • PaymentType	• Overly Optimistic

Source: The Willem Bogaerts Category (Copyright 2002 – 2006 Scott W. Ambler)

CONCURRENCY CONTROL AND LOCKING

Concurrency control and locking is the mechanism used by DBMSs for the sharing of data. Atomicity, consistency, and isolation are achieved through concurrency control and locking.

When many people may be reading the same data item at the same time, it is usually necessary to ensure that only one application at a time can change a data item. Locking is a way to do this. Because of locking, all changes to a particular data item will be made in the correct order in a transaction. The amount of data that can be locked with the single instance or groups of instances defines the granularity of the lock.

Model I - Mutual Exclusion (the ATM in mind)

As a simple example, consider a monitor for performing transactions on a bank account through an ATM machine.

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else if balance < amount then return false
    else { balance := balance - amount ; return true } }
    public method deposit(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else balance := balance + amount}}
```

While a thread is executing a method of a monitor, it is said to *occupy* the monitor. Monitors are implemented to enforce that *at each point in time, at most one thread may occupy the monitor*. This is the monitor's mutual exclusion property.

Upon calling one of the methods, a thread must wait until no thread is executing any of the monitor's methods before starting execution of its method. Note that without this mutual exclusion, in the present example, two threads could cause money to be lost or gained for no reason; for example two threads withdrawing 1000 from the account could both return without error while causing the balance to drop by only 1000.

In a simple implementation, mutual exclusion can be implemented by the compiler equipping each monitor object with a private lock, often in the form of a semaphore. This lock is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

Waiting and Signaling

For many applications, mutual exclusion is not enough. Threads attempting an operation may need to wait until some assertion *P* holds true. A busy waiting loop

```
  while not( P ) do skip
```

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true.

The solution is **condition variable**. Conceptually a condition variable is a queue of threads, associated with a monitor, upon which a thread may wait for some assertion to become true. Thus each condition variable *c* is associated with some assertion P_c. While a thread is waiting upon a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable *c* to indicate that assertion P_c is true.

Now fully in an ATM scenario, the user is expected to insert an ATM card. This card, first is validated to ascertain its expiration, next, the user's PIN is expected and also authenticated to be sure it is the right PIN. The account type and the amount to be withdrawn are next etc. All these are condition variables whose assertion must be true for the next condition to be executed. Thus there are two main operations on conditions variables:

- **wait** c is called by a thread that needs to wait until the assertion P_c to be true before proceeding.
- **signal** c (sometimes written as **notify** c) is called by a thread to indicate that the assertion P_c is true.

<u>Model II</u> - Blocking condition variables (the ATM in mind)

The original proposals by (Hoare (1974) and Hansen (1975) were for *blocking condition variables*. Monitors using blocking condition variables are often called *Hoare style* monitors. With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition.

We assume there are two queues of threads associated with each monitor object

- e is the entrance queue
- s is a queue of threads that have signaled.

In addition we assume that for each condition *c*, there is a queue

- *c*.q, which is a queue for threads waiting on condition *c*

All queues are typically guaranteed to be fair (i.e. each thread that enters the queue will not be chosen an infinite number of times) and, in some implementations, may be guaranteed to be first-in-first-out. The implementation of each operation is as follows. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

```
enter the monitor:
  enter the method
  if the monitor is locked
    add this thread to e
    block this thread
  else
    lock the monitor
leave the monitor:
  schedule
  return from the method
wait c :
  add this thread to c.q
  schedule
  block this thread
signal c :
  if there is a thread waiting on c.q
    select and remove one such thread t from c.q
    (t is called "the signaled thread")
    add this thread to s
    restart t
    (so t will occupy the monitor next)
    block this thread
schedule :
  if there is a thread on s
    select and remove one thread from s and restart it
    (this thread will occupy the monitor next)
  else if there is a thread on e
    select and remove one thread from e and restart it
    (this thread will occupy the monitor next)
  else
    unlock the monitor
    (the monitor will become unoccupied)
```

The schedule routine selects the next thread to occupy the monitor or, in the absence of any candidate threads, unlocks the monitor. The resulting signaling discipline is known a *"signal and urgent wait,"* as the signaler (the next most qualified thread to occupy) must wait, but is given priority over threads on the entrance queue. An alternative is *"signal and wait,"* in which there is no **s** (signaled threads) queue and signaler waits on the **e** (entrance) queue instead.

Some implementations provide a **signal and return** operation that combines signaling with returning from a procedure.

```
signal c and return :
  if there is a thread waiting on c.q
    select and remove one such thread t from c.q
    (t is called "the signaled thread")
    restart t
    (so t will occupy the monitor next)
  else
    schedule
  return from the method
```

In either case ("signal and urgent wait" or "signal and wait"), when a condition is signaled and there is at least one thread on waiting on the condition, the signaling thread hands occupancy over to the signaled

thread seamlessly, so that no other thread can gain occupancy in between. If P_c is true at the start of each **signal** *c* operation, it will be true at the end of each **wait** *c* operation. This is summarized by the following contracts. In these contracts, *I* is the monitor's invariant.

enter the monitor:
  postcondition *I*
leave the monitor:
  precondition *I*
wait *c :*
  precondition *I*
  modifies *the state of the monitor*
  postcondition P_c *and I*
signal *c :*
  precondition P_c *and I*
  modifies *the state of the monitor*
  postcondition *I*
signal *c* ***and return*** *:*
  precondition P_c *and I*

It is important to note here that the assertion P_c is entirely up to the programmer; he or she simply needs to be consistent about what it is.

Model III - Non-blocking condition variables (the ATM in mind)

With *nonblocking condition variables* (also called *"Mesa style"* condition variables or *"signal and continue"* condition variables), signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the e queue. There is no need for the s queue because the e (entrance queue) takes its place.

With nonblocking condition variables, the **signal** operation is often called **notify** - a terminology we will follow here. It is also common to provide a **notify all** operation that moves all threads waiting on a condition to the e queue. The meaning of various operations are given here. (We assume that each operation runs in mutual exclusion to the others; thus restarted threads do not begin executing until the operation is complete.)

enter the monitor:
  enter the method
  if the monitor is locked
    add this thread to e
    block this thread
  else
    lock the monitor
leave the monitor:
  schedule
  return from the method
wait *c :*
  add this thread to c.q
  schedule
  block this thread
notify *c :*
  if there is a thread waiting on c.q
    select and remove one thread t from c.q
    (t is called "the notified thread")
    move t to e
notify all *c :*
  move all threads waiting on c.q to e
schedule :
  if there is a thread on e
    select and remove one thread from e and restart it
  else
    unlock the monitor

As a variation on this scheme, the notified thread may be moved to a queue called w, which has priority over e. (John, 1976; Buhr *et al*., 1995).

It is possible to associate an assertion P_c with each condition variable *c* such that P_c is sure to be true upon return from **wait** *c*. However, one must ensure that P_c is preserved from the time the **notify**ing thread gives up occupancy until the notified thread is selected to re-enter the monitor. Between these times there could be activity by other occupants. Thus it is common for P_c to simply be *true*.

For this reason, it is usually necessary to enclose each **wait** operation in a loop like this

while not(*P*) **do wait** c

where *P* is some assertion stronger than P_c. The operations **notify** *c* and **notify all** *c* operations are treated as "hints" that *P* may be true for some waiting thread. Every iteration of such a loop past the first represents a lost notification; thus with non-blocking monitors, one must be careful to ensure that too many notifications can not be lost.

As an example of "hinting" consider a bank account in which a withdrawing thread will wait until the account has sufficient funds before proceeding

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0
  private NonblockingCondition balanceMayBeBigEnough
  public method withdraw(int amount)
  { if amount < 0 then error "Amount may not be negative"
    else { while balance < amount do wait balanceMayBeBigEnough
      assert balance >= amount balance := balance - amount } }
  public method deposit(int amount)
{ if amount < 0 then error "Amount may not be negative"
    else {balance := balance + amount
      notify all balanceMayBeBigEnough }  } }
```

In this example, the assertion being waited for is a function of the amount to be withdrawn, so it is impossible for a depositing thread to

be sure that it has established the assertion. So we can allow each waiting thread into the monitor (one at a time) to check if its assertion is true.

Model IV - Implicit condition monitors (the ATM in mind)

In the Java programming language each object may be used as a monitor. (However, methods that require mutual exclusion must be explicitly marked as **synchronized**). Rather than having explicit condition variables, each monitor (i.e. object) is equipped with a single wait queue, in addition to its entrance queue. All waiting is done on this single wait queue and all **notify** and **notify all** operations apply to this queue.

This approach has also been adopted in other languages such as C#.

CONCLUSION

There are by this work, four ways to reduce the number of database interactions. If we have say three requests to a database – the initial lock, (marking if appropriate) the source data, and unlocking; can both be performed as a single transaction. Thus two interactions say as to lock and obtain a copy of the source data, can easily be combined as a single trip to the database. Furthermore updating and unlocking can similarly be combined. Another way to improve this is to combine the last four models into a single transaction and simply perform collision detection on the database server instead of the application server.

The banking system has an interest in the protection of its financial assert to the detriment of the user in the ATM case. Here the bank is not too concern if the money requested for by the user was actually dispensed or not. But automatically debits the user's account. Considering the public method below; the first ***if*** structure considers whether the ATM user's account is in debit, since withdrawal is not permitted if in debit (red). Next, a check is made to verify if the requested amount for withdrawal is less than the ATM user bank's balance. Finally, clearance is given to the method as true for withdrawal to take place.

Once this is true, we propose a lock to the actual payment process which involves the evaluation of the new bank balance, invocation of a monitor to ascertain the fact that the cash has been dispensed

(- **Model II** - using a Blocking Conditioned Variable).

The initial concurrency control scheme was based on S-X locks for "flat," non-overlapping data objects. In order to adjust this scheme for practical applications, a set of concurrency control rules is derived for generalized lock modes described by a compatibility matrix.

Also, these rules are combined with a hierarchical locking scheme to improve selective access to data granules of varying sizes. After having tied together both types of hierarchies (transaction and object), it can be shown how "controlled downward inheritance" for hierarchical objects is achieved in nested transactions.

Finally, problems of deadlock detection and resolution in nested transactions have been considered. The critical region of any transaction must guard both parties and give protection to the object granules of both the user and that of the bank.

REFERENCES

Acken, K.P.; Irwin M.J., Owens R.M. (July 1998). A Parallel_ASIC Architecture for Efficient Fractal Image Coding. *The Journal of VLSI Signal Processing*, **19**(2):97–113(17)

Asanovic, Krste, et al. (2006). The Landscape of Parallel Computing Research: A View from Berkeley (PDF). University of California, Berkeley. Technical Report No. UCB/EECS-2006-183. :17–19.

Ben-Ari, (2006). *Principles of Concurrent and Distributed Programming* (2nd ed.). Addison-Wesley. NY.

Brinch Hansen, P. (1975). The programming language Concurrent Pascal. IEEE Trans. Softw. Eng. **2** :199–206.

Brinch Hansen, P. (1993). Monitors and concurrent Pascal: a personal history, *The second ACM SIGPLAN conference on History of programming languages* . Also published in *ACM SIGPLAN Notices* **28(3)**: 1 - 35

Buhr, P.H; Fortier, M., Coffin, M.H. (1995). Monitor classification. ACM Computing Surveys (CSUR) **27(1)** : 63–107.

Culler, David E.; Jaswinder Pal Singh and Anoop Gupta (1999). *Parallel Computer Architecture - A Hardware/Software Approach*. Morgan Kaufmann Publishers, p. 15. ISBN 1558603433.

Hoare, C. A. R. (1974) Monitors: an operating system structuring concept, Comm. A.C.M. **17(10) :** 549–57. [1]

John Howard (1976) Signaling in monitors. *Proceedings of the 2nd International Conference on Software Engineering*, 47–52

Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman(1987). *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, London.

Scott W. Ambler (2010). Introduction to Concurrency Control www.agiledata.org: Techniques for Successful Evolution/ Agile Database Development. http://www.agiledata.org/essays/concurrencyControl.html.

Taubenfeld, G. (2006). *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice Hall. NY, : 433.

Wilson, G.V (1994). The History of the Development of Parallel Computing. Virginia Tech/Norfolk State University, Interactive Learning with a Digital Library in Computer Science. http://ei.cs.vt.edu/~history/Parallel.html.