Towards a Unified Framework and Theory for Reasoning about Security and Correctness of Transactions in Multilevel Databases

Roshan K. Thomas and Ravi S. Sandhu[1]

Center for Secure Information Systems &
Department of Information and Software Systems Engineering
George Mason University
Fairfax, Virginia 22030-4444, USA

Abstract
The development of transaction management schemes is essential to the maturing of database technology for multilevel secure environments. Accordingly, several concurrency control and transaction management schemes have appeared in the recent literature. However, a close examination of these proposals reveal that they are cast in the context of individual problems and specialized architectures. Our objective in this paper is **not** to present yet another concurrency control (and transaction management) scheme; but rather to develop a unified framework and theory of multilevel transaction management from first principles. The long term vision is to develop a framework and theory that will be applicable for traditional as well as emerging (complex) transaction models.

Our approach is based on analyzing the various dependencies that can develop between transactions and database objects across security levels, leading to a better understanding of the crucial interplay between security and correctness. To guarantee correctness, any transaction processing scheme must schedule and complete transactions according to the commit order imposed by such dependencies. Hence, by utilizing these dependencies as the logical starting point, we are able to separate the analysis of insecurity arising from the maintenance of the correctness of transactions (concurrency control), from other causes. We present the groundwork for a theory of non-interference for concurrent multilevel transactions, that identifies commit dependencies between low and high level transactions as the central cause of interference. The presence of such dependencies will inevitably lead to a tradeoff between security and correctness. We demonstrate the applicability of our framework by analyzing the security and correctness of several existing proposals for transaction management.

Keyword Codes: H.2.0; K.6.5
Keywords: Database Management; Security and Protection

1. INTRODUCTION

A historical perspective on database management systems would attribute their usefulness, and much of their success, to the evolution of the *transaction* concept. The notion of a trans-

[1]The work of both authors was partially supported by the National Security Agency through contract MDA904-92-C-5140. We are indebted to Pete Sell, Howard Stainer and Mike Ware for making this work possible.

action as introduced by the early researchers in the field [4], considers it to be an atomic unit of execution. Hence, a transaction is considered to be both the unit of concurrency as well as the unit of recovery in applications. As a unit of concurrency, the concurrent execution of transactions causes no interference between transactions. This characteristic is often referred to as *serializability*, since the concurrent execution of transactions produces the net effect of a serial execution. As a unit of recovery, a transaction is characterized by the property of *failure atomicity* meaning that if a transaction succeeds, it performs all of its operations; if it fails it leaves the database unchanged.

The above concept of a transaction is appropriate for what we now call traditional database applications (such as relational systems processing inventory, billing or payroll), and has indeed served the database community well. Such database applications are characterized by transactions that are of short durations, and competing to access database objects. There exists a wide-body of knowledge addressing the theory of serializability and related transaction management issues [2].

However, this traditional view of transactions does not mesh well with the requirements and characteristics of databases for emerging application areas such as computer-aided design (CAD), office information systems, software development environments, cooperative work, etc. Why is this? Unlike traditional applications, transactions in these new areas (referred to as complex transactions) are often of long durations, and cooperative in nature. Hence, interactions and visibility across such transactions have to be promoted rather than curtailed, and serializability as the correctness criterion needs to be relaxed. Thus, there was a recognition that the transaction concept itself had to be reexamined [3,16], as it combined several important notions such as visibility, permanence, recovery, and consistency.[2]

The need for more flexible transaction models has resulted in various extensions and proposals to the traditional transaction model over the last decade [1,12]. As is perhaps inevitable, these initial efforts have a rather ad hoc flavor. This has led some researchers to reexamine transaction models and management issues from first principles, in an effort to gain a better understanding of the fundamentals at play, and to formulate a more unifying framework. The ACTA framework reported in [3] is a step in this direction. This framework allows us to capture the effects of transactions on other transactions, as well as the effect of transactions on other objects in the database, in terms of the dependencies that can develop between. The framework also allows us to deal with notions of visibility, permanence, recovery, and consistency with more individualized control. By utilizing dependencies as a common thread, it is able to accommodate traditional as well as emerging transaction models.

As researchers in database security, we believe there is wisdom and foresight in the adage: "*history often repeats itself*." Most research efforts in transaction management for multilevel secure database management systems (mls DBMS's) are still in their infancy. The various solutions and proposals in the literature are addressed to very specific problems under specialized architectures, and as such have been approached from fairly narrow considerations [6,7,10,13–15,17]. One is always tempted, by the excitement and prospects of unexplored territory, to pursue yet another algorithm or solution. Alternatively, one could reflect on lessons learned from the research experience in the non-secure database domain, and work towards developing paradigms and theories with more general applicability; thus, contributing to a better overall

[2]Visibility, refers to how one transaction during its lifetime, can see the effects of another. Permanence, refers to the ability of a transaction to record its results in the database permanently. Recovery, refers to the ability, as a consequence of a failure, to restore the database to some state that is considered to be correct. Consistency, pertains to the correctness of the database state that is produced by a committed transaction.

understanding of secure and correct transaction management.[3] We feel that the increasing interest (and maturity) of this area warrants a serious attempt at the latter approach.

Our main objective in this paper is to present the initial results of efforts aimed at developing a unified framework and theory for understanding security and integrity of transaction management in MLDBMS's.[4] We were led to this effort by our examination of the growing literature on this subject, and the subsequent difficulty in answering some fundamental questions such as the following.

- What is the interplay between security and the correctness (in terms of preserving consistency of the database) in transaction management?
- Is there a unifying approach to reasoning about the correctness and security of multilevel transactions?
- Is it possible to reason about the security of transaction management solutions irrespective of the underlying architecture and implementation?
- Can we formulate frameworks and theories to reason about the security and correctness of existing solutions, as well as non-traditional and emerging transaction models?

Our work has been influenced by the ACTA framework [3], mentioned above. Although ACTA has no relation to security, its flexibility to model traditional as well as complex transactions in a unifying manner, is appealing. Hence we utilize some of its key insights. In particular, we consider the dependencies formed by the interaction of transactions with other transactions, as well as objects, in the database. However, we bring in the additional dimension of security to the forefront by considering the implications of mandatory security rules on the formation of such dependencies. From these dependencies, we lay the groundwork for advancing a theory of non-interference for transactions executing at multiple security levels. We demonstrate how the formation of commit dependencies between transactions across security levels (low and high as well as incomparable) is the central cause of interference. We further analyze some existing transaction management schemes for the formation of such commit dependencies, and discuss their security.

To the best of our knowledge, the only other research effort to date, in developing a framework for analyzing the security of concurrency control and scheduling protocols is the one reported by Keefe and Tsai in [9–11]. Our work differs from that of Keefe and Tsai in many respects. Firstly, our work is based more on a bottom-up, first principles approach, and thus forms the substratum upon which the work of Keefe and Tsai (logically) rests. We illustrate, later in the paper, how the various conditions causing insecurity as identified by Keefe and Tsai can be reduced to interfering commit dependencies within our theoretical framework. Secondly, the Keefe and Tsai framework is molded too strongly by the traditional notions of concurrency control and transaction processing. On the other hand, our framework is built upon the dependencies that develop between transactions, and can thus be applied to the analysis of traditional as well as non-traditional concurrency control schemes and transaction models (such as various semantic concurrency control schemes and cooperative transaction models).

Our long-term vision is to develop a framework and theory for understanding and reasoning about security, integrity, and availability requirements, across the entire spectrum of transaction

[3]In this paper we use the term security, in the narrow sense of confidentiality or secrecy. The term correctness is used in the sense of internal consistency of the database (which is an aspect of integrity).

[4]When we refer to transaction management in this paper, we are specifically addressing the issue of concurrency control. We do not deal with other issues such as recovery.

management schemes. Hence it would have to be flexible enough to accommodate traditional as well as emerging (complex) transaction models. In this paper, we have limited most of our investigations to just traditional transaction models. Analysis of cooperative and other advanced (complex) transaction models is left for future work. We also limit the scope of our investigations to the interplay between security and integrity (correctness). Integration of availability issues would be too ambitious at this point, given the limited understanding of availability (denial-of-service) in transaction management.

The rest of this paper is organized as follows. Section 2 gives reviews the formation of dependencies among classical single-level transactions. Section 3 explores the basics of how dependencies can develop across transactions at multiple security levels. Section 4 investigates the vital link between dependencies and interference among transactions. Section 5 analyzes some well known architectures, and corresponding concurrency control proposals, for their security and correctness characteristics. Section 6 highlights some future directions for research, and concludes the paper.

2. TRANSACTIONS AND DEPENDENCIES

As transactions execute, they issue operations which access and modify database objects. When transactions execute concurrently, we have to consider the following:

1. the effect of transactions on each other, and

2. the effect of transactions on objects.

These two considerations provide the guiding light to building any broad and unifying framework to specify and analyze the entire spectrum of transaction management schemes.

The effects of concurrent transactions have implications on the correctness of the database, as well as on the security of the system as a whole. Hence, concurrent executions are governed by some correctness criterion (such as serializability, for traditional transaction processing). As far as security is concerned, the execution of high level transactions should not introduce (observable) interference from high-level to low-level users.[5] In this section we give a brief review of the correctness issue. We defer discussion on security until the next section.

It is useful to begin with a classification of the different types of operations in a database. In general, we classify an operation as an observer (O), modifier (M), or modifier-observer (MO). A read operation would be considered an observer, while a write would be a modifier. That is, a modifier changes the state of the object accessed, whereas an observer does not. The modifier-observer (MO) category is required to model abstract operations with semantics beyond primitive reads and writes. An operation of type MO, observes the value of an object, before modifying it. For example, an object that models a bank account would support operations such as deposit, withdraw, and balance. The balance operation is an observer, while the deposit operation is a modifier. The withdraw operation, on the other hand, is a modifier-observer; as it needs to check if the amount requested for withdrawal, exceeds the current balance in the account. Concurrent operations from different transactions conflict, if they access the same object and one of them is a modifier.

[5]For convenience, we loosely use the term user as synonymous with the traditional notion of subject in multilevel systems. In reality, of course, a high user may have low-level subjects acting on the user's behalf (when the user is logged in as low). Therefore, strictly speaking, we need to prevent interference from high-level subjects to low-level subjects. Also, we use only two levels, high and low, with high dominating low, in our examples. The framework, however, does apply to general lattices.

Operation 1 (p)	Operation 2 (q)
M	O
M	MO
MO	O
MO	MO

$\Big\}\ q \xrightarrow{\text{ad}} p$

Figure 1. The formation of abort dependencies

Operation 1 (p)	Operation 2 (q)
O	M
O	MO
MO	M
M	M

$\Big\}\ q \xrightarrow{\text{cd}} p$

Figure 2. The formation of commit dependencies

The effect of concurrent transactions, and their interacting operations, can be understood by examining the dependencies that develop between the transactions. Such dependencies impose a commit order among the various transactions and thus govern how they can be scheduled and completed. There are basically two types of dependencies, as mentioned in [3], and we define them below:

- **Commit-Dependency:** If a transaction t_1 develops a commit-dependency on a second transaction t_2 (denoted by $t_1 \xrightarrow{cd} t_2$), then t_1 cannot commit until t_2 terminates (i.e., either commits or aborts).

- **Abort-Dependency:** If a transaction t_1 develops an abort dependency on another transaction t2 (denoted by $t_1 \xrightarrow{ad} t_2$), then t_1 would have to abort whenever t_2 aborts.

Abort and commit dependencies—collectively known as **completion dependencies**—respectively arise between transactions due to **information flow** and **information obsolescence** that occur between them. If a transaction is abort-dependent on another, the information used by it would no longer be valid if the other transaction aborts. If a transaction A is commit-dependent on another transaction B, then A cannot effect its changes until B commits. This enforces serializability by ensuring that the information produced/used by B is not made obsolete.

Please note that the above dependencies are formed by transactions interacting over a shared object. Dependencies can also arise due to the structural properties of transactions (especially in advanced transaction models). For example, in a nested transaction model, a parent may not be able to commit until all its child (component) transactions have committed.

We elaborate on the formation of completion dependencies by considering various operation pairs (sequences), as shown in figures 1 and 2. Consider abort dependencies first. As can be seen in the first row in figure 1, a transaction (p) issuing a modifier operation followed by a second transaction (q) issuing an observer operation (the operation pair (M, O)), makes the second transaction abort dependent on the first. The operation pairs (M, MO), (MO, O), and (MO, MO) similarly cause abort dependencies to arise. In each of these cases, we notice that there is information flow from the first transaction to the second, because the operation in the second transaction observes the results produced by the first.

Commit dependencies are caused by the operation pairs (O, M), (O, MO), (MO, M), and (M, M), as shown in figure 2. In each case we notice that the second transaction with the modifier operation can overwrite (and thus obsolete) the value of the object accessed by the first transaction. To avoid this, the second transaction should delay its commit until the first transaction has committed. Otherwise, the first transaction will use the obsolete value, causing inconsistencies in the database.

We note there is no entry in the tables in figures 1 and 2 for the operation pair (O, O). This is because two concurrent observer operations will never conflict. Hence, they can be ordered in any fashion without causing dependencies to develop.

3. THE FORMATION OF DEPENDENCIES IN mls DBMS'S

We now explore the formation of completion dependencies among transactions executing at various security levels. We consider transactions that are classified at a security level, and whose operations are subject to mandatory security rules (as applied to that security level). Thus to begin with, we note that dependencies, across security levels, can form between transactions only as a result of conflicts arising through read-down and write-up operations (as read-up and write-down operations are prohibited by mandatory security rules in the Bell-LaPadula style access control policies). We consider the write-up case separately for ease of analysis.

We are particularly interested in what effect dependencies have on security. Consider what happens when a low-level transaction develops a dependency on a higher level transaction. In the case of a commit dependency, the low transaction cannot commit until the high one commits. If this can be modulated by the high transaction, we have the potential for a signaling channel. On the other hand, if a low transaction is abort dependent on a higher one, it would have to abort whenever the high transaction aborts. This could again lead to a signaling channel. Fortunately, it turns out (as we show in the analysis below) that such abort dependencies cannot form among multilevel transactions. Hence we have to be concerned only about commit dependencies.

3.1. Completion dependencies without write-up

We begin by looking at which operations are permitted by transactions on objects, by virtue of their security levels and classifications, respectively. As shown in the table in figure 3, an operation issued by a transaction may be an observer (O), modifier (M) or modifier-observer (MO), to an object at the same level. Without write-up, there is no operation compatibility between a low level transaction and a high level object. An operation from a high-level transaction, however, may be an observer (O) to a low level object.

The tables in figures 4 and 5 depict how completion dependencies form across transactions at multiple levels. These tables are derived directly by combining compatible (permissible) operations (as given in figure 3) with all completion dependencies generated by transactions in general (as given in figures 1 and 2, respectively for commit and abort dependencies). The operation pairs (MO, O) and (M, O) lead to high transactions developing abort dependencies on lower level transactions. This poses no security risk, and for analysis purposes can be largely ignored in this paper.[6] However, it is interesting to note that (even without write-up operations) the pairs (O, MO) and (O, M) can lead to low transactions becoming commit dependent on

[6]It should be noted that these dependencies can lead to denial of service problems for high-level transactions. A complete theory which addresses security, consistency and freedom from starvation would need to consider

(Trans, Object)	Object$_{lo}$	Object$_{hi}$
Trans$_{lo}$	O, MO, M	–
Trans$_{hi}$	O	O, MO, M

Figure 3. Operation compatibility matrix with no write-up

p$_{lo}$[Object$_{lo}$]	q$_{hi}$[Object$_{lo}$]
MO	O
M	O

$\Big\} \; \mathrm{t_{hi}} \xrightarrow{\mathrm{ad}} \mathrm{t_{lo}}$

Figure 4. Formation of abort dependencies with no write-up

p$_{hi}$[Object$_{lo}$]	q$_{lo}$[Object$_{lo}$]
O	MO
O	M

$\Big\} \; \mathrm{t_{lo}} \xrightarrow{\mathrm{cd}} \mathrm{t_{hi}}$

Figure 5. Formation of commit dependencies with no write-up

higher level ones. From a security perspective, these dependencies cannot be ignored as they could cause interference to low level transactions (as we will demonstrate in section 4).

3.2. Completion dependencies with write-up

When write-up operations are allowed, additional dependencies (than in the case with no write-up) form. To see this, consider the operation compatibility matrix in figure 6. In comparison to the matrix in figure 3 (for the no write-up case), we see that there is an extra M entry. This is because an operation issued by a low transaction may now be a modifier (M) to a high level object. The occurrence of such modifiers leads to the formation of the dependencies shown in figures 7, 8, and 9 (in addition to those of figures 4 and 5). The operation pairs (M, O) and (M, MO) are generated when a write-up operation by a low transaction precedes operations that are observers or modifier-observers from a high transaction. This causes the high transaction to be abort dependent on the low transaction. On the other hand, a high transaction becomes commit dependent on a lower one, when a write-up precedes a modifier operation from a high transaction (operation pair (M, M)).

The operation pairs (O, M), (M, M), and (MO, M)—generated when operations from high transactions precede write-up operations from a low transaction—lead to low transactions becoming commit-dependent on higher ones (as shown in the table in figure 9). Also, allowing write-up operations would lead to commit dependencies developing between transactions at incomparable levels. This is illustrated in the table in figure 10. Here we have two transactions at incomparable levels *la* and *lb* writing-up on a data item classified at a level *lab* that is an upper bound to *la* and *lb*. The resulting (M, M) operation sequence would lead to the transaction at

these dependencies. In this paper we have put aside the availability issue for the moment, so as to first unify confidentiality and consistency. Integration of availability into this framework is left for future work.

lb developing a commit dependency on the transaction at *la*. Again, from a security standpoint, these commit dependencies warrant further investigation. In particular, these dependencies have broad implications on concurrency control algorithms that allow write-up actions. In our further discussions and examples, we only consider dependencies between low and high transactions. However, all our results apply equally well to transactions at incomparable levels.

3.3. A Basic Property

We conclude this section by proving the fact that abort dependencies from low to high, as well as incomparable levels, cannot develop with multilevel transactions, as stated in the following theorem.

Theorem 1. In scheduling transactions at multiple levels, dependencies of the form $t_{la} \xrightarrow{ad} t_{lb}$ **cannot** develop, where t_{la} and t_{lb} are any two transactions at levels la and lb in the security lattice, such that $la \not\geq lb$.

Proof. For abort dependencies to develop, we need operation pairs of the form (M, O), (M, MO), (MO, O), or (MO, MO). In any one of these pairs, the result of a modifier (M) operation is made visible to a second observer (O) operation. Now for a low-level transaction to become abort dependent on a higher one, it would have to observe the modifications made by the higher-level transaction. However, since high transactions cannot write-down, and low transactions cannot read-up, this can never happen. That is, information cannot flow from high to low, and consequently abort dependencies, cannot develop. Similar arguments can be made for transactions at incomparable levels.□

From a theoretical standpoint, the absence of such abort dependencies is an important result for security and correctness considerations.[7] In terms of security, it means that the abortion of high transactions cannot induce (or modulate) the aborts of transactions at lower levels. Thus interference due to abort dependencies cannot occur. This allows us to narrow the scope of interference analysis, to just commit dependencies.

From a correctness perspective, the absence of abort dependencies implies that the abort of a high transaction cannot lead to a chain of cascading aborts at lower levels, for correctness reasons. Thus if the transactions at individual levels avoid cascading aborts (ACA), and there exists no high to low abort dependencies (i.e., no high transaction is abort dependent on a low transaction), the entire set of transactions across all security levels will also have the property of ACA. Histories that are ACA have very desirable properties such as recoverability [2].

4. COMMIT DEPENDENCIES AND INTERFERENCE

Having discussed the formation of low to high commit dependencies, we now discuss the implication of such dependencies on security and correctness. We first illustrate, by examples, how low to high commit dependencies can cause interference from higher to lower transactions (subjects). The only way to get around such interferences would be to ignore the dependencies. But doing so would clearly compromise correctness. Hence in the presence of such commit dependencies, there always exists a tradeoff between security and correctness. This fact has strong implications for any transaction management solution for multilevel systems.

[7]This property is also significant for availability, because it tells us that abortion of high transactions cannot induce denial-of-service or starvation of low transactions.

(Trans, Object)	Object$_{lo}$	Object$_{hi}$
Trans$_{lo}$	O, MO, M	M
Trans$_{hi}$	O	O, MO, M

Figure 6. Operation compatibility matrix with write-up

p$_{lo}$[Object$_{hi}$]	q$_{hi}$[Object$_{hi}$]
M	O
M	MO

$\Big\}\ t_{\mathrm{hi}} \xrightarrow{\mathrm{ad}} t_{\mathrm{lo}}$

Figure 7. Abort dependencies when low operation with write-up precedes high operation

p$_{lo}$[Object$_{hi}$]	q$_{hi}$[Object$_{hi}$]
M	M

$\Big\}\ t_{\mathrm{hi}} \xrightarrow{\mathrm{cd}} t_{\mathrm{lo}}$

Figure 8. Commit dependencies when low operation with write-up precedes high operation

p$_{hi}$[Object$_{hi}$]	q$_{lo}$[Object$_{hi}$]
O	M
M	M
MO	M

$\Big\}\ t_{\mathrm{lo}} \xrightarrow{\mathrm{cd}} t_{\mathrm{hi}}$

Figure 9. Commit dependencies when high operation precedes low operation with write-up

p$_{la}$[Object$_{lab}$]	q$_{lb}$[Object$_{lab}$]
M	M

$\Big\}\ t_{\mathrm{lb}} \xrightarrow{\mathrm{cd}} t_{\mathrm{la}}$

Figure 10. Commit dependencies between transactions at incomparable levels

4.1. Commit dependency as a source for interference

The approach of non-interference [5] to multilevel security has a strong appeal, due to its ability to abstract away unnecessary implementation details and artifacts. In this section we focus on developing the groundwork for the advancement of a theory of non-interference for multilevel transactions. Our hope is that such a theory would provide a useful tool for analyzing and predicting the security of existing as well as future transaction management schemes.

Interference from high to low-level transactions manifests itself in several ways. For example when high and low transactions are interleaved, it may be possible for a high-level transaction to modulate the value (of a low-level object) read by a low transaction (see example 1 below). In other scenarios, a low level transaction could experience interference by operations being delayed, or rejected, or by the entire transaction being aborted. Keefe and Tsai have categorized schedules which are free from such interferences as being value-secure, delay-secure, and recovery secure [9,10]. We illustrate below these, as well as other, scenarios. In each case, we observe that low transactions become commit dependent on high transactions (ie., a low to high commit dependency develops). In the following schedules, read, write, and commit operations are denoted with the letters R, W, and C, respectively.

Example 1: Interference through modulation of low reads

We illustrate this form of interference with an example taken from [10]. Consider the three transactions accessing an item x with the classifications and schedules as shown below:
High: T_1
Low: T_2, T_3, x

Schedule$_1$			
T_1:		R[x]	
T_2:			W[x]
T_3:	R[x]		

Schedule$_2$ = purge(Schedule$_1$, hi)		
T_2:	W[x]	
T_3:		R[x]

Schedule$_2$ is obtained by purging the Schedule$_1$ of the high transaction T_1. As a consequence of this purge, we see that T_3 reads a different value of x in Schedule$_2$ (one that is modified by the W[x] operation of T_2). Observe that the original schedule (Schedule$_1$) with the high transaction T_1, leads to the low transaction T_2 developing a commit dependency on the high transaction due to the presence of the operation sequence (R[x], W[x]) (which is of the form (O, M)). If the dependency is not followed, the value of x obtained by the R[x] operation of the high transaction T_1, will be inconsistent (as it will be obsolete). On the other hand, if the dependency is obeyed so as to guarantee consistency, the high transaction clearly causes interference (since it can modulate the value read by the lower transaction, as shown in Schedule$_2$). Let us see why. The possible serial orders for Schedule$_1$ include $T_3T_1T_2$ and $T_1T_3T_2$. On the other hand, the equivalent serial order of Schedule$_2$ is T_2T_3 (i.e., the order between T_2 and T_3 is now reversed). That is, in Schedule$_2$, the W[x] operation of T_2 could be placed ahead of the R[x] of T_1, thus giving the effect of T_2 committing before T_3. But to do the same in Schedule$_1$, that is making T_2 commit before T_3, implies that T_2 commit before T_1 as well. In summary, the presence of a low to high commit dependency in Schedule$_1$ prevents the scheduler from guaranteeing that such a schedule would be non-interfering if the high transaction had been purged (as in Schedule$_2$). If non-interference can be guaranteed, it can be done so only by compromising consistency.

Example 2: Interference through delays in scheduling

Here we are interested in how low-level transactions can experience interference through observable delays in the scheduling, and subsequent execution, of one or more operations. We say that an operation p_1 experiences a delay if it arrives before another operation p_2 (in the input schedule), but is scheduled and executed after p2 (in the output schedule). In considering how delays can cause interference, we are only interested in operations that conflict. This is because schedulers have the freedom to reorder nonconflicting operations according to the dominates relationship in the security lattice, thereby eliminating any delays.

If we look at the conflicting operation pairs given in the tables of figures 4, 5, 7, 8, and 9, we observe that only in figures 5 and 9 do operations from low transactions follow operations of higher transactions. These are the operation pairs (O, MO), (O, M), (M, M), (MO, M). But these operation pairs are precisely those that cause low to high commit dependencies to develop. In other words, if a low operation arrives before an operation from a higher transaction but is scheduled (in the output schedule) after the operation from the higher transaction, this will manifest in the output schedule as a low to high commit dependency. Conversely, if the output schedule is free of commit-dependencies, then we can conclude that all low operations (from lower level transactions) precede the corresponding conflicting operations from high transactions.

Example 3: Interference through delays in synchronization

In addition to scheduling delays, an operation may be delayed due to synchronization with conflicting operations from other transactions. Consider the example below with dynamic two-phase locking:

Schedule$_5$					
T_1:	lock[x]	R[x]		lock[y]	W[y]
T_2:			W[x]		

Transaction T_1 has obtained a lock on item x and proceeds with the R[x] operation. Hence the W[x] operation of the lower level transaction T_2 will be delayed (suspended) until this lock is released, leading to interference. To test for such interference, all we have to do is inspect the schedule for any low to high commit dependencies. Why? Because if a low transaction is never delayed waiting for a lock, the only conflicts that are possible, are those with observer operations from higher transactions. These result in the operation pairs (MO, O) and (M, O), as shown in the table in figure 4. But these pairs can cause only abort dependencies. On the other hand, whenever the high transaction acquires a lock first on a low item, the resulting operation pairs are of the form (O, MO), and (O, M), which as shown in the table in figure 5, cause low to high commit dependencies.

Example 4: Interference through rejected operations

In this example, transactions are timestamped on arrival. The lower transaction T_2 arrives first, and is thus assigned a smaller timestamp than the higher level transaction T_1. In other words T_2 is before T_1 in the equivalent serial order. Consider the schedule below. Although transaction T_1 arrived later than T_2, we assume that T_1 issued its first operation before T_2.

Schedule$_6$			
T_1:	R[x]		W[y]
T_2:		W[x]	

A timestamp based scheduler will reject the W[x] operation of the lower-level transaction T_2, as it has already processed the R[x] operation from a transaction with a larger timestamp; and the W[x], if allowed to proceed, would invalidate the value obtained by the R[x] operation. Such rejections of low operations can cause interference. Again this problem manifests itself in the commit dependency that the low transaction T_2 develops on the higher transaction T_1 as a result of the R[x] W[x] sequence. Avoiding such dependencies would eliminate such interferences.

Example 5: Interference through deadlocks

In this example we illustrate how interference is caused by deadlocks. Consider the following schedule where transactions use a dynamic two-phase locking protocol to obtain locks for accessing data items (these can be read or write locks).[8] Transaction T_1 is at a higher level than T_2.

<u>Schedule$_7$</u>

T_1: rlock[x]	r[x]		request-r-lock[y]	
T_2: wlock[y]	w[y]	request-w-lock[x]		

The above schedule leads to a deadlock as each transaction is requesting a lock held by the other. What role does low to high commit dependencies play in such a situation? Since low to high abort dependencies cannot develop across transactions at multiple levels, a cyclical wait-for relationship due to conflicting actions at multiple security levels (which is necessary for a deadlock) cannot exist unless there is a low to high commit dependency. In the schedule above, the write lock (request request-w-lock) of T_2 follows the r[x] operation of T_1, leading to the potential for the low transaction T_2 to develop a commit dependency on the higher transaction T_1. In other words, the potential for low to high commit dependencies to develop is indeed a necessary condition for a deadlock (the actual dependency develops only if the operation, associated with the requested lock, is allowed to proceed). In summary, low to high commit dependencies are necessary for interference through deadlocks across security levels.

4.2. Insecurity of Serializable Schedules

A well established and suitable correctness criterion for traditional transaction processing is *serializability*. As mentioned earlier in the introductory section, the traditional notion of a transaction is that it is an atomic unit, and hence indivisible. Serializability follows from the atomicity requirement. This is because if transactions are atomic units, then running them concurrently should produce the same effect as running them one after the other (serially) in isolation. To be more precise, a history (schedule) H is *serializable*, if it is equivalent to a serial history H_s. Equivalence here requires that the two histories have the same operations, and conflicting operations have the same ordering in both histories. A serializable history can be characterized by the following:

- There exists no cyclical dependencies of the form:
  $t_i \xrightarrow{d} t_j$ and $t_j \xrightarrow{d} t_i$ where d $\in$ {ad, cd}.

As an illustration, consider the following histories (operations are subscripted to identify the transaction issuing the operation).

S_a : W_1[x] R_2[x] R_2[y] W_1[y]

S_b : R_1[x] W_2[x] R_2[y] W_1[y]

Both schedules are not serializable. In S_a we have the dependencies $t_2 \xrightarrow{ad} t_1$ (due to W_1[x] preceding R_2[x]) and $t_1 \xrightarrow{cd} t_2$ (due to R_2[y] preceding W_1[y]). In S_b on the other hand, we have

[8]In dynamic two-phase locking, locks are acquired as required. However, with static two phase locking, a transaction is required to acquire all the necessary locks before it can proceed, and is thus deadlock free.

the dependencies $t_2 \xrightarrow{cd} t_1$ (due to $R_1[x]$ preceding $W_2[x]$) and $t_1 \xrightarrow{cd} t_2$ (due to $R_2[y]$ preceding $W_1[y]$) . If such cycles in the dependencies exist, there is no equivalent serial history (in this case for S_a and S_b).

The inadequacy of serializability theory for reasoning about the security of multilevel transactions stems from the fact that it addresses only the correctness issue. In particular, the specification for serializability does not preclude the presence of low to high commit dependencies among transactions at various security levels. Further, the equivalent serial order, and the order in which transactions actually commit, could be different. To put it differently, the assertion that a transaction processing system (scheduler) guarantees serializability, says nothing about its security properties.

We elaborate on the above with multiversion schedules (histories). The serializability based correctness criterion for multiversion histories is called *one-copy serializability.* We say a multiversion history is one-copy serializable if it is equivalent to a 1-serial multiversion history. In 1-serial histories, operations of individual transactions are not interleaved and a read operation always obtains the last written version. Consider the multiversion schedule below, and its 1-serial equivalent (version numbers are indicated by subscripts attached to the data items).

$\underline{\text{Schedule}_{mv}}$

T_1 (hi):				$R_1[X_3]$	
T_2 (lo):	$R_2[Y_0]$				$W_2[X_2]$
T_3 (lo):		$R_3[Z_0]$	$W_3[X_3]$		

$S_{1-serial}$:	$R_2[Y_0]$	$W_2[X_2]$	$R_3[Z_0]$	$W_3[X_3]$	$R_1[X_3]$

The history is clearly one-copy serializable and transaction $t_{3,lo}$ precedes $t_{1,hi}$ in the serial order. However, there is a low to high commit dependency from $t_{2,lo}$ to $t_{1,hi}$ since the operation $W_2[X_2]$ invalidates the value read by operation $R_1[X_3]$.

In summary, it is thus important to note that serializability and related theory are artifacts of an era of database development where correctness alone was the overriding concern. With the advent of multilevel secure databases, there is clearly a need to reexamine such theories. Any criterion to govern transaction processing in the security context, has to incorporate both security and correctness in a unified manner. This will enable us to reason about the security as well as correctness of transaction management solutions, and further provide the foundation for developing a systematic methodology to design such solutions. Our approach in this paper of identifying commit dependencies as a cause for interference is the first step in understanding the link (interplay) between security and correctness.

5. ANALYSIS OF VARIOUS ARCHITECTURES AND TRANSACTION MANAGEMENT SCHEMES

In this section, we analyze some of the well known architectural approaches and solutions that have been reported in the literature. We discuss the implications of low to high commit dependencies in trusted[9] subject, kernelized, and replicated architectures. In the replicated

[9]The term "trusted" is used often in the literature to convey one of two different notions of trust. In the first case, it conveys the fact that something is trusted to be correct. In the second case, we mean that some subject is exempted from mandatory confidentiality controls; in particular the simple-security and $\star$-properties in the Bell-Lapadula framework. It is the latter sense of trust that we refer to in this paper. In general, a trusted subject is expected not to violate mandatory information flow eventhough it is exempted from mandatory access controls.

architecture, the backend databases are untrusted. Any trusted subject, would have to reside in the trusted front end (TFE). The discussion on the architectures is necessary as they have an impact on the type of schedulers (concurrency controllers) that can be implemented, and it is the schedulers that have to enforce the various dependencies.

Rather than overwhelming the reader with comprehensive details and terminologies of each of these solutions, we have opted to be concise and present the crux of the matter at hand. In other words, how do these solutions handle commit dependencies and the related interference problem?

We begin with some definitions to characterize clearly the notions of correctness, as well as freedom from low to high commit dependencies.

Definition 1. We define a transaction management system (scheduler) to be **correctness preserving** if given any set of of transactions T, transactions commit according to the order consistent with commit and abort dependencies that develop when executing T.

Definition 2. We define a transaction management system to be **low-dependency free** if given any set of transactions T, the system can guarantee that no dependencies of the form $t_1 \xrightarrow{cd} t_2$ can develop, where t_1 and t_2 are transactions at levels l_1 and l_2 respectively, and $l_1 \not\geq l_2$.

The rest of this section is organized as follows. In sections 5.1, 5.2 and 5.3, we discuss the broad implications of commit dependencies in the replicated, trusted subject, and kernelized architectures respectively. This leads us to a categorization of systems and implementations in section 5.4, based on their concurrency control characteristics. Section 5.5 gives an analysis of existing proposals for concurrency control in MLDBMS's, relating these proposals to the categorization of section 5.4.

5.1. Commit dependencies in replicated architectures

We explain (and formalize) an interesting scenario involving commit dependencies, that arises with transaction processing in architectures supporting replicated data. In such systems, a separate database management system is used to manage data at or below each security level, and also contains all the data at its level or below. Thus lower level data is replicated across all databases containing higher level data. When a low level transaction updates its local copy, it is required to send an update report (or a replica of the transaction itself) to a trusted front end (TFE). The TFE subsequently distributes these reports to other containers (databases) at higher levels that hold copies of the same data item. In this way, the mutual consistency of the copies (replicas) is ensured.

In some of the proposals for replica control, the TFE raises the level of the incoming update report (or transaction replica) to a level say, l, before distributing it to the database at l. We refer to such update reports whose levels have been raised, as *clones* of the original parent transaction. (A clone transaction will only contain the modifier (write) operations issued by its parent.) Now at first glance, it seems that no low to high commit dependency can develop at level l (since the clone and any local incoming transaction at l would be at the same level). However, the clone developing a commit dependency on the local transaction is semantically equivalent to the original parent transaction developing a dependency on the local transaction at l, if they both access the same data item. In some sense, the dependency permeates through the clone. We define a cloning operator, using the notation $\models$, to express this. For example, $t_1 \models t_2$, conveys the fact that a lower transaction has created a clone at a higher level. The notation $t_{lo} \models tc_{hi} \xrightarrow{cd} t_{hi}$ indicates a commit dependency between the low transaction t_{lo} and the higher transaction t_{hi}, after the creation of the clone transaction tc.

5.2. Commit dependencies in trusted subject architectures

Architectures that utilize trusted subjects are characterized by the fact that such subjects are exempt from mandatory security rules. From a transaction processing viewpoint, the advantage of using trusted subjects is that it makes it feasible to implement trusted (multilevel) schedulers. Such subjects can access information (data structures) at various levels, and maintain a global snapshot of transactions across multiple levels, as they progress. A low transaction can now be informed of the commit of a higher one with a write-down operation by a trusted subject, enabling the low to high commit order to be enforced. Thus the maintenance of correctness appears to be an achievable goal. But the following conjecture poses a dilemma:

Conjecture 1. If a transaction management system is implemented under an architecture with trusted subjects, and is not low-commit-free, it will compromise security to be correctness preserving. (Of course, it may not be correctness preserving and still compromise security.)

The problem with trusted subject architectures is that non-interference arguments have to be made to demonstrate the security of the system. However this is not an easy task. There are always unexpected scenarios one may overlook, and often it is difficult to identify let alone cover all cases. Even if interference does not manifest itself in an obvious fashion, such as when transactions are delayed or rejected, other forms of interference are inevitable. This could happen for example, due to the finite nature of resources in the system, leading to data structures getting saturated. In some architectures with trusted subjects, it might be possible for the system to guarantee security. But this can be done only by ignoring the commit order mandated by the dependencies. Correctness is then clearly compromised.

5.3. Commit dependencies in kernelized architectures

The difficulties in evaluating trusted subject architectures have sparked an increased interest in kernelized architectures. In these architectures, all subjects are single level and thus untrusted (if a subject is trusted, it is exempt from mandatory security rules). Security comes for free in such architectures as there is no information flow downwards in the security lattice. Hence, at first sight, it might appear that our fear of security violations from interference can be allayed by architectural solutions. But consider the following conjecture:

Conjecture 2. A transaction management subsystem implemented under a kernelized architecture and which is not low-dependency-free (i.e., allows low to high commit dependencies to develop), can guarantee security (as it comes for free), but not correctness (as it may not be correctness preserving).

Now it is not possible for the commit time of a high transaction to be made known to a dependent low transaction, without any subject having the privilege to write down. Single-level (untrusted) subjects and scheduler components do not have such privileges (as they are not exempt from mandatory security access control rules). Thus in a system that is not low-dependency free, it would not be possible to schedule transactions according to the order imposed by the commit dependencies. There also exist other scenarios where a kernelized architecture may not be able to keep up with incoming requests (transactions), forcing it to eventually compromise correctness.

5.4. A categorization of systems and implementations

Conjectures 1 and 2 collectively point to the fundamental conflict between security and correctness. Any architecture that allows low to high commit dependencies to develop, would have to settle for a tradeoff between security and correctness, to a certain degree. A third implementation alternative would be to pursue solutions that place restrictions on how transactions can be interleaved, so that commit dependencies can be avoided (these systems are thus inherently low dependency free).

We are thus led to believe that any transaction processing system and the associated implementation can be classified as belonging to one of the three categories below:

- Category A: Systems which are low-dependency free.
- Category B: Systems which are not low-dependency free. We recognize two sub-cases.
  - Category B_1: Systems which do not use trusted subjects for concurrency control.
  - Category B_2: Systems which use trusted subjects for concurrency control.

(Recall that a trusted subject is exempted from the simple-security and star-properties, whereas an untrusted subject must operate under these constraints.)

We believe that any category A system can be implemented under both trusted subject and kernelized architectures. The porting of such a system from a trusted subject to a kernelized architecture should not pose any significant difficulty as it basically involves the modification of centralized coordination tasks, with algorithms that enable distributed coordination. For example, global schedulers may now have to be be implemented as distributed single-level schedulers. This has been undertaken in some of the systems presented in [13,17]. Of course, the benefit of doing this would be that the assurance of security now comes for free. As mentioned before, category B_1 systems will never compromise security (to be more precise confidentiality) but this assurance comes at the heavy price of correctness. Category B_2 systems, in the worst case, may compromise both security and correctness.

5.5. Analysis

In this section we present some of the well known proposals in the growing pool of transaction management schemes that have been reported in the literature. In presenting them, we identify the category (A, B_1, or B_2) to which each of them belong.

5.5.1. Solutions for the Replicated Architecture

Jajodia and Kogan: Category B_2

We have classified this as a category B_2 solution, since the architecture utilizes a trusted (subject) front end (TFE) [6]. Hence by conjecture 1, this solution will not be able to simultaneously guarantee security and correctness. Let us see if our claim holds up. The fundamental issue in this architecture is the maintenance of the mutual consistency of the replicas, in a secure fashion. For simplicity, consider just two levels, high and low, and one data item x (with two replicas, one at each level). Once a low level transaction t_{low} commits, its clone, $t_{clone,_{high}}$ (update projection) is dispatched to the high database by the the TFE. However, on reaching the high database, $t_{clone,_{high}}$ finds that there is a local transaction $t_{local,_{high}}$ that has read data item x and is still active (not committed). Due to the resulting commit dependency (which can happen with both locking and timestamped based intra-level synchronization) the commit of the clone (update projection) will be held up until the local transaction commits. Recall from the last section, that this is a low to high commit dependency of the form $t_{low} \models t_{clone,_{high}} \xrightarrow{cd} t_{local,_{high}}$.

At first glance, this poses no interference (insecurity) since, as argued in [6], the commit of the parent transaction is never held up by the clone. Unfortunately, a closer examination yields evidence to the contrary. We need to see if the trusted subject in the architecture, which in this case is the TFE, can cause interference (perhaps unintentionally) in other ways. Consider what happens if the update projections of a million low-level transactions are sent to the TCB. (This is rather an extreme scenario, but illustrates the point.) The TCB stores these projections in a queue (which has a finite capacity). Now if the commit of the original update projection (clone) has still not been processed, the queue in the TFE cannot be emptied (serviced), and will soon overflow as it will be unable to keep up with the high rate of incoming update projections. The TFE now has two options: (1) to reject or delay further incoming update projections; or (2) to still accept further projections at the cost of discarding others that are overflowing from the head of the queue. Option 1 causes interference as the observable effect of the system's behavior to low level subjects would be different had the commit dependency not developed (i.e., if $t_{local,hi}$ had been purged from the system). On the other hand, with option 2, the system attempts to provide non-interfering behavior, but can succeed only by forsaking correctness (as it does not service all the update projections, thereby affecting the mutual consistency of the replicas of x). In either case conjecture 2 is true and our claim holds.

It is important to stress that the above interference scenario arises precisely because of the low to high commit dependency, and not due to design flaws in the TFE. The design of the TFE could be an ideal one, with the queue capacity tuned to accommodate the highest (worst case) input rates of update projections. However, the above scenario would still break such an ideal design, so long as the capacity of the queue is finite. It is also significant to note that interference may manifest itself in other observable ways, such as system response time, and other performance characteristics, to name a few.

Kang, Froscher, Costich: Category B$_2$

This solution is one that uses trusted subjects [8], and is thus vulnerable to the same scenario illustrated for the earlier proposal of Jajodia and Kogan. It utilizes a tree model of transactions. A set of local schedulers are concerned with the scheduling of the lower layer primitive read and write actions, while a global (trusted) scheduler is responsible for the scheduling of more abstract higher layer operations (queries). Unfortunately low to high commit dependencies will develop under the scheduling strategies of the local schedulers, leading to interference.

McDermott, Jajodia, and Sandhu: Category B$_1$

In this approach to replica control, single-level (untrusted) conservative timestamp ordering schedulers are used [14]. Low to high commit dependencies may still develop as in the previously illustrated scenario. This could lead to one or more of the single-level queues at the back-end databases becoming saturated, and consequently correctness cannot be guaranteed. However, as there are no trusted subjects, the security of the system (non-interference) is never compromised.

5.5.2. Multiversioning Schemes

Keefe and Tsai: Category A.

In [10], Keefe and Tsai describe a multiversion timestamp ordering (MVTO) algorithm with an implementation that calls for a trusted subject. It is claimed to be secure and can handle write-up operations. Let us see why. The basic intuition behind their approach can be seen by considering the following simple schedules.

Schedule$_{8a}$

T_{1a} (high):	R[x]	
T_{2a} (low):		W[x]

Schedule$_{8b}$

T_{1b}:	R[x] Commit$_{1b}$	
T_{2b}:		W[x]

In Schedule$_{8a}$, if the lower transaction T_{2a} is assigned an earlier timestamp than the higher transaction T_{1a}, the W[x] operation of the lower transaction would invalidate the higher R[x] operation, and this obviously causes a low to high commit dependency to develop. The scheduler avoids these situations by assigning to the higher transaction T_{1a}, a lower timestamp than T_{2a}. Hence semantically, transactions execute as in schedule$_{8b}$, with the higher transaction committing well before the W[x] operation can invalidate its read. In summary a commit dependency is avoided.

Maimone and Greenberg: Category A.

This is basically an implementation of the Keefe and Tsai approach using single-level schedulers [13]. As mentioned earlier, category A systems can always be ported to architectures which don't support trusted subjects. In this revised implementation, timestamps are obtained from a shared clock at system low. Each scheduler maintains two values, the earliest timestamp (ETS), and the earliest lower timestamp (ELTS), to implement a timestamping scheme in a distributed fashion.

Jajodia and Atluri: Category A.

The approach presented by Jajodia and Atluri in [7] is also based on multiversion timestamp ordering, and uses single-level schedulers. It can be described as a category A solution, and hence avoids commit dependencies. However the approach different from that of Keefe and Tsai. If a high level read is found to be invalidated by a lower write operation, the high transaction is rolled back to the read operation and re-executed. Also, the commit of the high transaction is delayed until the lower transaction has committed. In effect, a schedule such as Schedule$_{8a}$ above, is transformed to the one below:

Schedule$_{8c}$

T_{1c} (high):			R[x]
T_{2c} (low):	W[x]	Commit$_{2c}$	

We see that the low to high commit dependency in Schedule$_{8a}$ is eliminated by making sure that the lower transaction with the W[x] operation has committed well before the high read. The Keefe, Tsai and Jajodia, Atluri schemes represent two extreme approaches in eliminating (avoiding) interfering low to high commit dependencies. In the former, livelocks are avoided at the price of high transactions being given older versions of data (than those chosen with classical timestamping schemes). In the latter, high transactions do not receive very old versions of data, but may have to rollback and restart.

5.5.3. Lock based approaches: Orange Locking: McDermott and Jajodia: Category A

Unlike the multiversion timestamp ordering schemes above, the approach in [15] is for single-version databases, and uses locking for concurrency control. Data items can be read locked or write locked. When a high level transaction wants to read lower data, it sets a read-(down) lock on the needed item. If a lower transaction wants to write a data item, it is unconditionally allowed to set a write lock and proceed. However, if a read-down lock is held by a higher transaction on this item just locked for writing, the read lock is changed to an *orange lock*.

The orange lock indicates the potential for a low to high commit dependency to develop (as we have an operation pair (R[x], W[x]) which is of the form (O, M)). The high transaction is then aborted and would have to reissue the read operation. It will be allowed to commit at some point when it has read locked all data items none of these locks are orange.

In summary, this approach is similar to that used in Schedule_{8_c} above, but is cast in terms of locking. The authors limit the scope of the algorithms in [15] to transactions that write only at their own levels. To handle write-up operations, the additional low to high commit dependencies that result from the operation pairs (M, M), (MO, M) in table 9, need to be considered. In this case, orange locks will have to be applied to write locks as well, since a lower write can invalidate a preceding write at a higher level.

6. SUMMARY AND CONCLUSIONS

In this paper, we have presented a unifying approach for reasoning about the security and correctness of transaction management in multilevel secure databases. The approach is based on analyzing the dependencies that develop between concurrently interacting transactions. While it is necessary to obey the commit order induced by such dependencies to guarantee correctness, our approach makes it easy to check if the maintenance of correctness itself can cause security violations. Thus in designing any transaction management scheme, our framework makes it easy to answer such questions as: (1) is the transaction management scheme secure? Is it correct? (2) In trying to guarantee both security and correctness, is one achieved at the expense (compromise) of the other?

Our investigations reveal that analyzing the presence or absence of commit dependencies between low and high transactions (or those at incomparable levels) is sufficient to give an answer to these questions. In particular, it is sufficient to find one schedule which has such dependencies to assert that the system cannot simultaneously achieve security and correctness. On the other hand, proving the absence of these dependencies establishes that the concurrency control component does not compromise security or correctness (Of course, security might still be compromised due to flaws in other components of the system).

We have demonstrated in this paper how some of the solutions proposed for transaction management in replicated architectures do not avoid such commit dependencies, and hence can cause interference under some scenarios. However, the elegance of our approach is that it eliminates guesswork and gives the designer a clear roadmap on how the problem can be fixed.

In this paper, we have limited the scope of our discussion to traditional transaction processing (including write-up). However abort and commit dependencies also develop due to transactions sharing objects through abstract operations defined in their interfaces. Hence by utilizing these dependencies as a common thread, we are in a position to extend our investigations to the security and correctness of complex transactions and concurrency control schemes (such as for object-oriented databases).

In our future work, we hope to advance the ideas presented here into a comprehensive framework that addresses all aspects of transaction processing in multilevel databases. In particular, the effect of dependencies on the availability (and denial of service) issue needs to be incorporated. We would also like to incorporate the idea of transaction authorization. This would allow authorization mechanisms to be based on the semantics of the transactions and the objects accessed.

REFERENCES

1. F. Bancilhon, W. Kim, and H. F. Korth. A model of CAD transactions. *Proc. of the VLDB conference*, 1985, Stockholm, pp. 25-33.
2. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading, Mass, (1987).
3. P. K. Chrysanthis and K. Ramamritham. "ACTA: A Framework for specifying and reasoning about transaction structure and behavior," *Proc. of the ACM SIGMOD conference*, pages 194–203, 1990.
4. J. Gray, "The Transaction concept: virtues and limitations," *Proc. of the Seventh International Conference on Very Large Databases, 1981.*
5. J. A. Goguen and J.Meseguer, "Security policy and security models." *Proc. IEEE Symp. on Research in Security and Privacy,* Oakland, Calif., May 1982, pages 11-20.
6. Sushil Jajodia and Boris Kogan, "Transaction processing in multilevel-secure databases using replicated architecture." *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, May 1990, pages 360–368.
7. Sushil Jajodia and Vijayalakshmi Atluri, "Alternative correctness criteria for concurrent execution of transactions in multilevel secure database systems," *Proc. IEEE Symp. on Research in Security and Privacy,* Oakland, Calif., May 1992, pages 216–224.
8. M. H. Kang, J. N. Froscher, and O. Costich, "A practical transaction model and untrusted transaction manager for a multilevel-secure database system," *Proc. of the IFIP 11.3 Workshop on Database Security*, Vancouver, Canada, August 1992.
9. T. F. Keefe and W. T. Tsai and J. Srivastava, "Database concurrency control in multilevel secure database management systems" Technical Report, TR 89-73, Computer Science Department, Univ. of Minnesota, Minneapolis MN 55455, 1989.
10. T. F. Keefe and W. T. Tsai, "Multiversion concurrency control for multilevel secure database systems." *Proc. IEEE Symp. on Research in Security and Privacy*, May 1990, pages 369–383.
11. T. F. Keefe, W. T. Tsai, and J. Srivastava, "Multilevel secure database concurrency control," *Proc. IEEE 6th Int'l. Conf. on Data Engineering,* Los Angeles, California, February 1990, pages 337–344.
12. W. Kim, R. Lorie, D. McNabb, and W. Plouffe. "A transaction mechanism for engineering design databases," *Proc. of the VLDB conference*, 1984, Singapore, pp. 355-362.
13. William T. Maimone and Ira B. Greenberg, "Single-level multiversion schedulers for multilevel secure database systems," *Proc. 6th Annual Computer Security Applications Conf.,* Tucson, Arizona, December 1990, pages 137–147.
14. J.P. McDermott, S. Jajodia, and R.S. Sandhu "A single-level scheduler for the replicated architecture for multilevel-secure databases." *Proc. 7th annual computer security applications conference,* San Antonio, Texas, pages 2–11.
15. John McDermott and Sushil Jajodia, "Orange locking: Channel-free database control via locking," *Proc. 6th IFIP WG 11.3 Working Conf. on Database Security,* Vancouver, British Columbia, August 1992.
16. A. H. Skarra and S. B. Zdonik, "Concurrency Control and Object-oriented databases," In *Object-Oriented Concepts, Databases, and Applications*, pages 395-421, ACM Press, 1989.
17. R.K. Thomas and R.S. Sandhu, "Implementing the message filter object-oriented security model without trusted subjects," *Proc. of the IFIP 11.3 Workshop on Database Security*, Vancouver, Canada, August 1992.