Essay 27

Integrity Mechanisms in Database Management Systems

Ravi S. Sandhu and Sushil Jajodia

Information integrity means different things to different people, and will probably continue to do so for some time. The 1989 NIST workshop, which set out to establish a consensus definition, instead arrived at the following conclusion [NIST89, page 2.6]:

> The most important conclusion to be drawn from this compilation of papers and working group reports: don't draw too many conclusions about the appropriate definition for data integrity just yet.... In the mean time, papers addressing integrity issues should present or reference a definition of integrity applicable to that paper.

So the first order of business is to define integrity. Our approach to this question is pragmatic and utilitarian. The objective is to settle on a definition within which we can achieve practically useful results, rather than search for some absolute and airtight formulation.

We define integrity[1]as being concerned with the *improper modification* of information (much as confidentiality is concerned with improper disclosure). We understand modification to include insertion of new information and deletion of existing information, as well as changes to existing information.

[1]Our definition of integrity is considerably broader than the traditional use of this term in the database literature. For instance, Date [DATE86] says, "Security refers to the protection of data against unauthorized disclosure, alteration, or destruction; integrity refers to the accuracy or validity of data." The consensus view among security researchers is that integrity is one component of security and accuracy/validity is one component of integrity [FERN81, NIST89].

The reader has probably seen similar definitions using "unauthorized" instead of "improper." Our use of the latter term is quite deliberate and significant. First, it acknowledges that security breaches can and do occur without authorization violations — that is, authorization is only one piece of the solution. Second, it adheres to the well-established and useful notion that information security has three components: integrity, confidentiality, and availability. We see no need to discard this standard viewpoint in the absence of some compelling demonstration of a superior one. Finally, our definition brings to the front the very important question: What do we mean by improper? It is obvious that this question intrinsically cannot have a universal answer. So it is futile to try to answer it outside of a given context.

We are specifically interested in information systems used to control and account for an organization's assets. In such systems, the primary goal is prevention of fraud and errors. The meaning of improper modification in this context has been given by Clark and Wilson [CLAR87] as follows:

> No user of the system, even if authorized, may be permitted to modify data items in such a way that assets or accounting records of the company are lost or corrupted.

Note their express qualification: "even if authorized." The word "company" in this quotation reveals the authors' commercial bias but, as they have clarified [CLAR89a], these concepts apply equally well to any information system that controls assets — be it in the military, government, or commercial sectors.

Our goal in this essay is to answer the following question: What mechanisms are required in a general-purpose multiuser DBMS to help achieve the integrity objectives of information systems? There are many compelling reasons to focus on DBMSs for this purpose. The most important has been succinctly stated by Burns [BURN89] as follows:

> A database management system (DBMS) provides the appropriate level of abstraction for the implementation of integrity controls as presented in the Clark and Wilson paper [CLAR87].... It is clear that the domain of applicability of the Clark and Wilson model is not an operating system or a network or even an application system, it is fundamentally a DBMS.

This is particularly true when we focus on mechanisms. Moreover, DBMSs have the wonderful ability to express and manipulate complex relationships. This comes in very handy when dealing with sophisticated integrity policies.

The operating system must clearly provide some core integrity and security mechanisms. In terms of the Orange Book [DOD85], one would need at least a B1 system to enforce encapsulation of the DBMS — that is, to ensure that all manipulation of the database can only be through the DBMS. The question of what minimal features are required in the operating system is important, but outside the scope of this essay. For now, let us assume that operating systems with the requisite features are available.

The bulk of integrity mechanisms belong in the DBMS. Integrity policies are intrinsically application specific, and the operating system simply cannot provide the means to state application-specific concerns. One might then argue: Why not put all the mechanism in the application code? There are several persuasive reasons not to do this. First, it is not very conducive to reuse of common mechanisms. Second, any assurance that integrity mechanisms interspersed within application code will be correct or even comprehensible is rather dubious. Third, the whole point of a database is to support multiple applications. A particular application may well be in a position to handle all its integrity requirements. Yet it is only the DBMS which can prevent other applications from corrupting the database.

The rest of the essay is organized as follows. First we discuss principles for achieving integrity in information systems. Then we describe the mechanisms required in a DBMS to support these high-level principles. In some of the more detailed considerations, we will limit ourselves specifically to relational DBMSs. As we will see, traditional DBMS mechanisms provide the foundations for this purpose, but by themselves do not go far enough.

Integrity principles

We begin by describing basic principles for achieving information integrity. These can be viewed as high-level objectives that are made more concrete when specific mechanisms are proposed to support them. In other words, these principles lay down broad goals without specifying how to achieve them. We will subsequently map these principles to DBMS mechanisms. We emphasize that the principles themselves are independent of the DBMS context. They apply equally well to any information system, be it a manual paper-based system, a centralized batch system, an interactive and highly distributed system, and so on.

The nine integrity principles enumerated below are abstracted from the Clark and Wilson papers [CLAR87, CLAR89a, CLAR89b], the NIST workshops [NIST87, NIST89], and the broader security and database literature.[2]These principles express what needs to be done rather than how it is going to be accomplished (the latter question is addressed in the next section):

1. *Well-formed transactions.* Clark and Wilson [CLAR87] have defined this principle as follows: "The concept of the well-formed transaction is that a user should not manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of the data." This principle has also been called *constrained change* [CLAR89b] — that is, data can be modified only by well-formed transactions rather than by arbitrary procedures. Moreover, the well-formed transactions are known ("certified") to be individually correct with some (mostly qualitative) degree of assurance.
2. *Authenticated users.* This principle stipulates that modifications should be carried out only by users whose identities have been authenticated to be appropriate for the task.
3. *Least privilege.* The notion of least privilege was one of the earliest to emerge in security research. It has classically been stated in terms of processes (executing programs) [SALT75]: A process should have exactly those privileges needed to accomplish its assigned task, and none extra. The principle applies equally well to users, except that it is more difficult to precisely delimit the scope of a user's "task." A process is typically created to accomplish some very specific task and terminates on completion. A user, on the other hand, is a relatively long-lived entity and will be involved in varied activities during his life span. His authorized privileges will therefore exceed those strictly required at any given instant. In the realm of confidentiality, least privilege is often called *need-to-know.* In the integrity context, it is appropriately called *need-to-do.* Another appropriate term for this principle is *least temptation* — that is, do not tempt people to commit fraud by giving them greater power than they need.
4. *Separation of duties.* Separation of duties is a time-honored principle for prevention of fraud and errors, going back to the very beginning of commerce. Simply stated, no single individual should be in a position to misappropriate assets on his own. Operationally, this means that a chain of events that affects the balance of assets must require different individuals to be involved at key points, so that without their collusion the overall chain cannot take effect.
5. *Reconstruction of events.* This principle seeks to deter improper behavior by threatening its discovery. It is a necessary adjunct to

[2]The literature is too numerous to cite works individually. For those unfamiliar with the "older" literature, there are some useful starting points [DENN79, FERN81, GRAY78, LIND76, SALT75].

least privilege for two reasons. First, least privilege, even taken to its theoretical limit, will leave some scope for fraud. Second, a zealous application of least privilege is not a terribly efficient way to run an organization. It conveys an impression of an enterprise enmeshed in red tape.[3] So practically, users must be granted more privileges than are strictly required. We therefore should be able to accurately reconstruct essential elements of a system's history, so as to detect misuse of privileges.

6. *Delegation of authority.* This principle fills in a piece missing from the Clark and Wilson papers and much of the discussion they have generated.[4]It concerns the critical question of how privileges are acquired and distributed in an organization. Clearly, the procedures to do so must reflect the structure of the organization and allow for effective devolution of authority. Individual managers should have maximum flexibility regarding information resources within their domains, tempered by constraints imposed by their superiors. Without this flexibility at the end-user level, the authorization will most likely be inappropriate to the actual needs. This can only result in security being perceived as a drag on productivity and something to be bypassed whenever possible.
7. *Reality checks.* This principle has been well motivated by Clark and Wilson [CLAR89b] as follows: "A cross-check with the external reality is a central part of integrity control. ...integrity is meaningful only in terms of the relation of the data to the external world." Or in more concrete terms: "If an internal inventory record does not correctly reflect the number of items in stock, it makes little difference if the value of the recorded inventory has been reflected correctly in the company balance sheet."
8. *Continuity of operation.* This principle states that system operations should be maintained to some appropriate degree in the face of potentially devastating events beyond the organization's control. This catchall description is intended to include natural disasters, power outages, disk crashes, and the like.[5]

[3]This comment is made in the context of users rather than processes (transactions). Least privilege with respect to processes is more of an internal issue within the computer system, and its zealous application is most desirable (modulo the performance and cost penalties it imposes).

[4]The closest concept that Clark and Wilson have to this principle is their Rule E4, which they summarize as follows [CLAR87, Figure 1]: "Authorization lists changed only by the security officer." This notion of a central security officer as an authorization czar is inappropriate and unworkable. Rational security policies can be put in place only if appropriate authority is vested in end users.

[5]One might argue that we are stepping into the scope of availability here. If so, so be it.

9. *Ease of safe use.*[6] In a nutshell, this principle requires that the easiest way to operate a system should also be the safest. There is ample evidence that security measures are all too often incorrectly applied or simply bypassed by the system managers. This happens due to one or a combination of the following: (1) poorly designed defaults (such as indefinite retention of vendor-supplied passwords for privileged accounts), (2) awkward and cumbersome interfaces (such as requiring many keystrokes to effect simple changes in authorization), (3) lack of tools for authorization review, and (4) mismatched policy and mechanism ("...to the extent that the user's mental image of his protection goals matches the mechanism he must use, mistakes will be minimized" [SALT75]).

It is inevitable that these principles are fuzzy, abstract, and high level. In developing an organization's security policy, one would elaborate on each of these principles and make precise the meaning of terms such as "appropriate" and "proper." How to do so systematically is perhaps the most important question in successful application of these principles. In other words, how does one articulate a comprehensive policy based on these high-level objectives? This question is beyond the scope of this essay. Our present focus is on this question: How do these principles translate into concrete mechanisms in a DBMS?

The goals encompassed by these principles may appear overwhelming. After all, in the extreme these principles amount to solving the total system correctness problem, which we know is well beyond the state of the art. Fortunately, in our context, the degree to which one would seek to enforce these objectives and the assurance of this enforcement are matters of risk management and cost-benefit analysis. Laying out these principles explicitly does give us the following major benefits:

- The overall problem is partitioned into smaller components for which solutions can be developed independently of each other (that is, divide and conquer).
- The principles suggest common mechanisms that belong in the DBMS and can be reused across multiple applications.
- The principles provide a set against which the mechanisms of specific DBMSs can be evaluated (in an informal sense).
- The principles similarly provide a set on the basis of which the requirements of specific information systems can be articulated.
- Last, but not least, the principles invite criticism from the security community, particularly regarding what may have been left out.

[6]Thanks to Stanley Kurzban and William Murray for coining this term.

Integrity mechanisms

In this section, we consider DBMS mechanisms to facilitate application of the principles defined in the previous section. The principles have been applied in practice [MURR87a, WIMB71], but with most of the mechanisms built into application code. Providing these mechanisms in the DBMS is a prerequisite for their widespread use.

Our mapping of principles to mechanisms is summarized in Table 1. Some of these mechanisms are available in commercial products. Others are well established in the database literature. There are also some newer, recently proposed mechanisms, for example, transaction controls for separation of duties [SAND88b], the temporal model for audit data [JAJO90g], and propagation constraints for dynamic authorization [SAND88a, SAND89]. Finally, there are places where existing mechanisms and proposals need to be extended in novel ways. Overall, the required mechanisms are quite practical and well within the reach of today's technology.

Table 1. Summary.

Integrity Principle	DBMS Mechanisms
Well-formed transactions	Encapsulated updates Atomic transactions Consistency constraints
Continuity of operation	Redundancy Recovery
Authenticated users	Authentication
Least privilege	Fine-grained access control
Separation of duties	Transaction controls Layered updates
Reconstruction of events	Audit trail
Delegation of authority	Dynamic authorization Propagation constraints
Reality checks	Consistent snapshots
Ease of safe use	Fail-safe defaults Human factors

Well-formed transactions. The concept of a well-formed transaction corresponds very well to the standard DBMS concept of a transaction [GRAY78, GRAY86]. A transaction is defined as a sequence of primitive actions that satisfies the following properties:

1. *Failure atomicity.* Either all or none of the updates of a transaction take effect. We understand update to mean modification; that is, it includes insertion of new data, deletion of existing data, and changes to existing data.
2. *Serializability.* The net effect of executing a set of transactions is equivalent to executing them in some sequential order, even though they may actually be executed concurrently (that is, their actions are interleaved or simultaneous).
3. *Progress.* Every transaction will eventually complete; that is, there is no indefinite blocking due to deadlock and no indefinite restarts due to livelock.
4. *Correct state transform.* Each transaction if run by itself in isolation and given a consistent state to begin with will leave the database in a consistent state.

We will elaborate on these properties in a moment. First let us note the basic requirement that the DBMS must ensure that updates are restricted to transactions. Clearly, if users are allowed to bypass transactions and directly manipulate relations in a database, we have no foundation to build upon. We represent this requirement with the diagram in Figure 1 — updates are encapsulated within transactions. At this point it is worth recalling that the database itself must be encapsulated within the DBMS by the operating system.

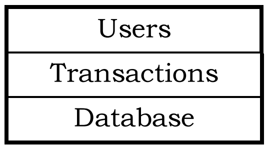

Figure 1. Encapsulated updates.

It is clear that the set of database transactions is itself going to change during the system life cycle. Now the same nine principles of the previous section apply with respect to maintaining the integrity of the transactions. In particular, transactions should be installed, modified, and supplanted only by the use of well-formed "transaction-maintenance transactions." One can apply this argument once again to say that the transaction-maintenance transactions themselves need to be maintained by another set of transactions, and so on indefinitely. We believe there is little to be gained by having more than two steps in this potentially unbounded sequence of transaction-maintenance transactions. The rate of change in the transaction set will be significantly slower than the rate of change in the database proper. Going one step further, the rate of change in the transaction-maintenance transactions will be yet slower to the point where, for all practical purposes, these can be viewed as static over the life span of typical systems. With this perspective, the database administrator is responsible for installing and maintaining transaction-maintenance transactions, which in turn maintain actual database transactions.

We now return to considering the four properties of DBMS transactions enumerated earlier. The first three properties — failure atomicity, serializability, and progress — can be achieved in a purely "syntactic" manner — that is, completely independent of the application. These three requirements for a transaction are recognized in the database literature as appropriate for the DBMS to implement. Mechanisms to achieve these objectives have been extensively researched in the last 15 years or so, and our understanding of this area can certainly be described as mature. The basic mechanisms — two-phase locking, time stamps, multiversion databases, two-phase commit, undo-redo logs, shadow pages, deadlock detection and prevention — have been known for a long time and have made their way into numerous products. In developing integrity guidelines and/or evaluation criteria, one might consider some progressive measure of the extent to which a particular DBMS meets these objectives. For instance, with failure atomicity, is there a guarantee that we will know which of the two possibilities occurred? Similarly, with serializability, does the DBMS enforce the concurrency control protocol or does it rely on transactions to execute explicit commands for this purpose? And, with the issue of progress, do we have a probabilistic or absolute guarantee? Such questions must be systematically addressed.

The fourth property, correct state transforms, is the ultimate bottleneck in realizing well-formed transactions. It is also an objective that cannot be achieved without considering the semantics of the application. The correctness issue is, of course, undecidable in general. In practice, we can assure correctness only to some limited degree of confidence by a mix of software engineering techniques such as formal verification, testing, quality assurance, and so on. Responsibility for implementing transactions as correct state transforms has traditionally been assigned to the application programmer. Even in theory, DBMS mechanisms can never fully take over this responsibility.

DBMS mechanisms can help in assuring the correctness of a state by enforcing *consistency constraints* on the data. Consistency constraints are also often called integrity constraints or integrity rules in the database literature. Since we are using integrity in a wider sense, we prefer the term consistency constraint.

The relational data model in particular imposes two consistency constraints [CODD79, DATE86]:

- *Entity integrity* stipulates that attributes in the primary key of a base relation cannot have null values. This amounts to requiring that each entity represented in the database must be uniquely identifiable.
- *Referential integrity* is concerned with references from one entity to another. A foreign key is a set of attributes in one relation whose values are required to match those of the primary key of some specific relation. Referential integrity requires that either a foreign key be all null[7]or a matching tuple exist in the latter relation. This amounts to ruling out dangling references to nonexistent entities.

Entity integrity is easily enforced. Referential integrity, on the other hand, requires more effort and has seen limited support in commercial products. The precise manner in which to achieve it is also very dependent on the semantics of the application. This is particularly so when the referenced tuple is deleted. There are several choices:

1. prohibit this delete operation,
2. delete the referencing tuple (with a possibility of further cascading deletes), or
3. set the foreign key attributes in the referencing tuple to null.

There are proposals for extending SQL so that these choices can be specified for each foreign key.

The relational model in addition encourages the use of *domain constraints*, whereby the values in a particular attribute (column) are constrained to come from some given set. These constraints are particularly easy to state and enforce, at least so long as the domains are defined in terms of primitive types such as integers, decimal numbers, and character strings. A variety of *dependency constraints* [DATE86] that constrain the tuples in a given relation have been extensively studied in the database literature.

In the limit, a consistency constraint can be viewed as an arbitrary predicate that all correct states of the database must satisfy. The predicate may involve any number of relations. Although this concept is theoretically appealing and flexible in its expressive power, in practice the overhead in checking the predicates for every transaction has been pro-

[7]Often the notion of a null foreign key is semantically incorrect. In such cases, an additional consistency constraint can disallow null values.

hibitive. As a result, relational DBMSs typically confine their enforcement of consistency constraints to domain constraints and entity integrity.

Continuity of operation. The problem of maintaining continuity of operation in the face of natural disasters, hardware failures, and other disruptive events has received considerable attention in both theory and practice [GRAY78]. The basic technique to deal with such situations is redundancy in various forms. Recovery mechanisms in DBMSs must also ensure that we arrive at a consistent state. In many respects, these mechanisms are “syntactic” in the sense of being application independent, much as mechanisms for the first three properties presented in the section “Well-formed transactions” were.

Authenticated users**.** Authentication is primarily the responsibility of the operating system. If the operating system is lacking in its authentication mechanism, it would be very difficult to ensure the integrity of the DBMS itself. The integrity of the database would thereby be that much more suspect. It therefore makes sense not to duplicate authentication mechanisms in the DBMS.

Authentication underlies some of the other principles, particularly least privilege, separation of duties, reconstruction of events, and delegation of authority. In all of these, the end objective can be achieved to the fullest extent only if authentication is possible at the level of individual users.

Least privilege. The principle of least privilege translates into a requirement for fine-grained access control. Earlier we noted that least privilege must be tempered with practicality in avoiding excessive red tape. Nevertheless, a high-end DBMS should provide for access control at very fine granularity, leaving it to the database designers to apply these controls as they see fit.

It is clear from the Clark and Wilson papers, if not evident from earlier work, that modification of data must be controlled in terms of transactions rather than blanket permission to write. We have already put forth the concept of encapsulated updates for this purpose. In terms of the relational model, it is not immediately obvious at what granularity of data this should be enforced.

To control read access, DBMSs have used mechanisms based on views (as in System R) or query modification (as in INGRES). These mechanisms are extremely flexible and can be as fine grained as desired. However, neither one provides the same potential for flexible control of updates. The fundamental reason for this is our theoretical inability to translate updates on views unambiguously into updates of base relations. As a result, authorization to control updates is often less sophisticated than authorization for read access.

In relational systems, it is natural for obvious reasons to represent the access matrix by one or more relations [SELI80]. At a coarse level, we might control access by tuples of the following form:

user, transaction, relation

This means that the specified user can execute the specified transaction on the specified relation. Tuples of the form shown below would give greater selectivity:

user, transaction, relation, attribute

This would allow us to control the execution of transactions such as "give everyone a 5 percent raise," without giving the same transaction permission to change employee addresses. The following authorization tuple accomplishes this:

Joe, Give-5%-raise, Employees, Salary

A transaction that gives a raise to a specific employee needs a further dimension of authorization to specify which employee it pertains to. Thus, if Joe is authorized to give a 5 percent raise to John, the authorization tuple would look as follows:

Joe, Give-5%-raise, John, Employees, Salary

We are assuming here that John uniquely identifies the employee receiving the raise. The update is restricted to the Salary attribute of a specific tuple with key equal to "John" in the Employees relation. So it takes a key, relation, and attribute to specify the actual parameter of such a transaction.

Now consider a transaction which moves money from account A to account B; that is, there are two actual parameters of the transaction. In terms of least privilege, we need the ability to bind this transaction to updating the two specific accounts A and B. More generally, we will have transactions with *N* parameters identified in an actual parameter list. So we need authorization tuples of the following form:

user, transaction, actual parameter list

Here each parameter in the actual parameter list specifies the item authorized for update by specifying one of the following identifiers:

- relation,
- relation, attribute,

- key, relation, attribute.

These three cases respectively give us relation-level, "column"-level, and element-level granularity of update control.

It is also important to realize that element-level update authorizations should properly be treated as consumable items. For example, once money has been moved from account A to account B, the user should not be able to move it again, without fresh authorization to do so.

Separation of duties. Separation of duties finds little support in existing products. Although it is possible to use existing mechanisms for this purpose, these mechanisms have not been designed with this end in mind. As a result, their use is awkward at best. This fact was noted by the DBMS group at the 1989 NIST data integrity workshop, who concluded their report with the following recommendation [NIST89, section 4.3]:

> While the group was able to use existing DBMS features to implement separation of roles controls, we were, however, unable to use existing features in a way that would support easy maintenance and certification. We recommend that data definition and/or consistency check features be enhanced to provide operators that lend themselves to the expression of integrity controls and to allow separation of integrity controls and traditional data.

Separation of duties is inherently concerned with sequences of transactions, rather than individual transactions in isolation. For example, consider a situation in which payment in the form of a check is prepared and issued by the following sequence of events:

1. A clerk prepares a voucher and assigns an account.
2. The voucher and account are approved by a supervisor.
3. The check is issued by a clerk who must be different from the clerk in step 1. Issuing the check also debits the assigned account. (Strictly speaking, we should debit one account and credit another in equal amounts. The important point for our purpose is that issuing a check modifies account balances.)

This sequence embodies separation of duties since the three steps must be executed by different people. The policy, moreover, has a dynamic flavor in that a particular clerk can prepare vouchers as well as, on different occasions, issue checks. However, he cannot issue a check for a voucher prepared by himself.

Implementation of this policy in a paper-based system follows quite directly from its statement:

- The voucher is realized as a form with blanks for the amount and account, as well as for signatures of the people involved. As the above sequence gets executed, these blanks are filled in. On its completion, copies of the voucher are filed in various archives for audit purposes.
- The account is represented by, say, a ledger card, where debit and credit entries are posted, along with references to the forms that authorized these entries.

By their very nature, paper-based controls rely on employee vigilance and internal/external audits for their effectiveness. Computerization brings with it the potential to enforce the required controls by means of an infallible, ever-vigilant, and omniscient automaton — the computer itself.

The crucial question is, how do we specify and implement similar controls for separation of duties in a computerized environment? A mechanism for this purpose called *transaction-control expressions* [SAND88b] is based on the following difference between vouchers and accounts:

- The voucher is *transient* in that it comes into existence, has a relatively small sequence of steps applied to it, and then disappears from the system (possibly leaving a record in some archive). The history of a voucher can be prescribed as a finite sequence of steps with an a priori maximum length.
- The account, on the other hand, is *persistent* in the sense that it has a long-lived — and essentially unbounded — existence in the system. During its life there may be a very large number of credit and debit entries for it. Of course, at some point the account may be closed and archived. The key point is that we can only prescribe its history as a variable-length sequence of steps with no a priori maximum length.

Both kinds of objects are essential to the logic and correct operation of an information system. Transient objects embody a logically complete history of transactions corresponding to units of service provided to the external world by the organization. Persistent objects embody the internal records required to keep the organization functioning with an accurate correspondence to its interactions with the external world.

Separation of duties is achieved by enforcing controls on transient objects, for the most part. The crucial idea that makes this possible is that transactions can be executed on persistent objects only as side effects of executing transactions on transient objects. This thesis is actually simply borrowed from the paper-based world, where it has been routinely applied ever since bookkeeping became an integral part of business operations.

With this perspective, we arrive at the diagram shown in Figure 2. The idea is that a sequence of transactions is viewed as transient data in the database. In this picture, there is a double encapsulation of the database, first by transactions on persistent data and then by transactions on transient data. Users can directly execute only the latter. The former are triggered indirectly as a result, when the transient data is in the proper state for doing so. In other words, transient data is singly encapsulated and has direct application of separation of duties. Persistent data is doubly encapsulated and has indirect application of separation of duties by means of transient data.

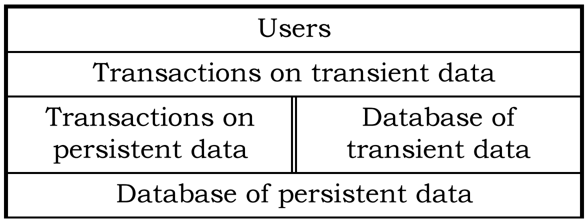
Users
Transactions on transient data
Transactions on persistent data
Database of transient data
Database of persistent data

Figure 2. Layered updates.

Reconstruction of events**.** The ability to reconstruct events in a system serves as a deterrent to improper behavior. In the DBMS context, the mechanism to record the history of the system is traditionally called an audit trail. As with the principle of least privilege, a high-end DBMS should be capable of reconstructing events to the finest detail. It should also structure the audit trail logically so that it is easy to query. For instance, logging every keystroke does give us the ability to reconstruct the system history accurately. However, with this primitive logical structure, it takes substantial effort to reconstruct a particular transaction. In addition to actually recording all events that take place in the database, an audit trail must also provide support for auditing. In other words, an audit trail must allow “an authorized and competent agent to access and evaluate accountability information by a secure means, within a reasonable amount of time and without undue difficulty” [DOD85]. In this respect, DBMSs have a significant advantage, since their powerful querying abilities can be used.

The ability to reconstruct events means different things to different people. At one end of the spectrum, we have the requirements of Clark and Wilson [CLAR89b]. They require only two things:

1. A complete history of each and every modification made to the value of an item.
2. With each change in value of an item, storage of the identity of the person making the change.

Of course, the system must be reliable in that it makes exactly those changes that are requested by users and the binding of a value with its author is also exact. Clark and Wilson call this "attribution of change."

This can be easily accomplished if we are willing to extend slightly the standard logging techniques for recovery purposes. For each transaction, a recovery log contains the transaction identifier, some *before-images*, and the corresponding *after-images*. If we augment this by recording the user for each transaction, we have the desired binding of each value to its author. There is one other change that needs to be made. To support recovery, there is a need to keep a log only up to a point from which a complete database backup is available. Of course, now there is a need to archive the logs so they remain available.

Others have argued that this simple "attribution of change" is not sufficient. We need an audit trail, a mechanism for a complete reconstruction of every action taken against the database: *who* has been accessing *what* data, *when*, and in what *order*. Thus, it has three basic objects of interest:

1. *The user.* Who initiated a transaction, from what terminal, when, and in what order?
2. *The transaction.* What was the exact transaction that was initiated?
3. *The data.* What was the result of the transaction? What were the database states before and after the transaction initiation?

For this purpose, a *database activity model* has been recently proposed [JAJO90g] that imposes a uniform logical structure upon the past, present, and future data. There is never any loss of historical or current information in this model; thus the model provides a mechanism for complete reconstruction of every action taken on the database. It also logically structures the audit data to facilitate its querying.

Delegation of authority. The capability to delegate authority and responsibility within an organization is essential to its smooth functioning. It appears in its most developed form with respect to monetary budgets. However, the concept applies equally well to the control of other assets and resources of the organization.

In most organizations, the ability to grant authorization is never completely unconstrained. For example, a department manager may be able to delegate substantial authority over departmental resources to project managers within his department and yet be prohibited to delegate this authority to project managers outside the department. These situations cloud the classic distinction between discretionary and mandatory policies [MURR87b, SAND90]. The traditional concept of ownership as the basis for delegating authority also becomes less applicable in this context [MOFF88]. Finally, we need the ability to delegate privileges without having the ability to exercise these privileges. Some mechanisms for this purpose have been recently proposed [MOFF88, SAND89].

The complexity introduced by dynamic authorization has been recognized ever since researchers considered this problem, for example, as stated by Saltzer and Schroeder [SALT75]:

> ...it is relatively easy to envision (and design) systems that statically express a particular protection intent. But the need to change access authorizations dynamically...introduces much complexity into protection systems.

This continues to be true, despite substantial theoretical advances in the interim [SAND88a]. Existing products provide few facilities in this respect, and their mechanisms tend to have an ad hoc flavor.

Reality checks**.** This principle inherently requires activity outside of the DBMS. The DBMS does have the obligation to provide an internally consistent view of that portion of the database which is being externally verified. This is particularly so if the external inspection is conducted on an ad hoc on-demand basis.

Ease of safe use. Ease of safe use is more an evaluation of the DBMS mechanisms than something to be enforced by the mechanisms themselves. The mechanisms should, of course, have fail-safe defaults [SALT75] — for example, access is not available unless explicitly granted or this default rule is explicitly changed to grant it automatically. DBMSs do offer a significant advantage in providing user-friendly interfaces intrinsically, for their main objective of data manipulation. These interface mechanisms can be leveraged to make the authorization mechanisms easy to use. For instance, having the power of SQL queries to review the current authorizations is a tangible benefit.

Conclusion

In a nutshell, our conclusion is that realistic DBMS mechanisms do exist to support the integrity objective of information systems. Some are well established in the literature, while others have been proposed more recently and are not so well known.

In terms of what DBMS mechanisms can do for us, we can group the nine principles enumerated in this essay as follows:

Group I	**Group II**	**Group III**
Well-formed transactions	Least privilege	Authenticated users
Continuity of operation	Separation of duties	Reality checks
	Reconstruction of events	Ease of safe use
	Delegation of authority	

Group I principles are adequately treated by current DBMS mechanisms and have been extensively studied by database researchers. With the single exception of assuring correctness of state transformations, these principles can be achieved by DBMS mechanisms. Techniques for implementing well-formed transactions and maintaining continuity of operation across failures have been studied extensively. Their practical feasibility has been amply demonstrated in actual systems. Assuring that well-formed transactions are correct state transformations remains a formidable problem, but there is little that the DBMS can do to alleviate it. As such, it is a problem outside the scope of DBMS mechanisms. The DBMS can

1. enforce encapsulation of updates by restricting their occurrence to be within transactions, and
2. provide controls for installing and maintaining these transactions.

Group II principles need newer mechanisms and conceptual foundations. Several promising approaches have emerged in the literature. Practical demonstration of their feasibility remains to be done, but in concept they do not present prohibitive implementation problems. They do require that current DBMSs be extended in significant ways. Group II principles are the ones for which additional DBMS mechanisms hold the promise of greatest benefit.

Group III principles are important, but there is little that DBMS mechanisms can do to achieve them. Authentication is principally an operating system problem. Reality checks necessarily involve external procedures. Ease of safe use is more an evaluation of the DBMS mechanisms than something to be enforced by the mechanisms themselves. It is facilitated in the DBMS context by the intrinsic DBMS requirement of user-friendly query languages.

In conclusion, for group I principles we need little more than has currently been demonstrated in actual products. For group II principles, current systems do something for each one but do not go far enough. There are several promising proposals but no “worked examples.” Group III principles are important but are not fully achievable by DBMS mechanisms alone.