# Consistency Issues in Real–Time Database Systems

Kwei–Jay Lin

*Department of Computer Science*
*University of Illinois at Urbana–Champaign*
*Urbana, Illinois 61801*

**Abstract**

To provide a transaction schedule to meet deadlines, we define a new consistency model of real–time database systems which distinguishes the *external* data consistency from the *internal* data consistency as maintained by traditional systems. External consistency requires that the data used by a transaction reflect the current physical environment; this is in contrast to internal consistency which presents a view consistent with the predefined constraints of the database. We suggest that external consistency is preferable to internal consistency for many transactions. We define *operationally consistent* schedules which emphasize the operational effect of databases to the external world. We also present a protocol which ensures the external consistency of transactions.

## 1. Introduction

As computers are used more to control and monitor complex real–life systems, many computations are required to satisfy real–time constraints. Yet we are only beginning to understand the actual implications of real–time constraints on computer systems [1,9,10]. Although many consider real–time issues to be simply performance issues which can be resolved by careful hand planning or automatic code optimization, we believe that real–time systems are a totally different generation of computer systems which call for a new discipline and technology. There needs to be a fundamental breakthrough in the basic computing theory and structure to facilitate a powerful and structured approach to build complex real–time systems.

To build complex real–time systems, we believe that systems should be designed with a different emphasis, i.e. meeting deadlines, right from the beginning. It is the objective of this paper to suggest one such new approach in designing real–time database systems. Traditionally, database systems are designed to provide only functionally correct information. Although databases are also concerned about providing adequate performance, the correctness of a database operation is hardly affected by the slowness of a transaction. In other words, conventional database models do not include deadline parameters. Very few, if any, query languages allow users to specify the timing constraints. Similarly, temporal databases have been designed to handle historical information in which time is an element of the data. But temporal databases assume that all information in the database is valid in that it can be used for processing a query regardless of age. In other words, the age factor does not change the usefulness of the data in the database. All of these are inappropriate for real–time databases which have transactions with timing constraints, and have a constant flow of data values each with a limited life span.

In this paper, we discuss issues in combining various consistency schemes. Examples are shown in which transactions have a new requirement of data consistency. Traditionally, the serialization property is maintained among transactions in order to ensure the data consistency in the database. However, in some applications, providing a timely response may require the consistency to be compromised. We discuss why this may be acceptable and intend to establish a new notion of correctness for real–time databases. In the next section, we discuss the rationale behind our model. We present the concept of internal and external consistency, and discuss some classes of real–time transactions in Section 3. Section 4 presents a scheduling protocol which facilitates the new consistency model.

## 2. Features of Real–Time Databases

The ability to meet all deadlines requested by all system events is vital to a real–time system. In real–time databases, transactions may be executed periodically or aperiodically. Periodic transactions are executed at constant intervals such as a metering transaction which enters the meter reading every minute. There can also be periodic transactions which maintain statistical information, like a transaction which produces the average reading for the hour. The deadline of a periodic

This work was supported in part by a contract from the ONR (N00014–87–K–0827).

transaction is usually defined to be the end of the current period, although earlier deadlines are also possible. Aperiodic transactions are usually triggered by predefined events. The deadlines for aperiodic transactions are individually defined. Some of the deadlines are *hard* deadlines which cannot be missed or the result is useless. Others are *soft* deadlines in which the usefulness of a result decreases as the delay after the deadline increases.

### 2.1. Functional vs. Timing Correctness

When traditional concurrency control protocols are applied to real–time applications, transactions may fail to complete before their deadlines because of the delay by concurrent activities. We are interested in the implication of real–time constraints and data consistency on the concurrency control issues of real–time databases. In certain situations, if a transaction does not have enough time to complete its execution, it may be able to produce incomplete, or just *imprecise*, results from its operations. To reduce the failure propagation due to aborted transactions, and to get the most out of otherwise wasted time and resources, it is desirable to make use of these imprecise results if possible. In our earlier research [7,8], we have shown that the imprecise computation model can be very useful. That model was mainly concerned with computations with no shared resources. In this paper, we extend the work in designing imprecise computations to transaction mechanisms in databases.

To facilitate timely executions which meet deadlines, we extend the definition of correctness in database transactions just like in imprecise computations. A new consistency model of real–time database systems called *external* data consistency is defined in contrast to *internal* data consistency as maintained by traditional systems. The external consistency constraint requires that the data used by a transaction reflect the physical environment at the time; this is in contrast to internal consistency which requires that all data must meet some predefined constraints in the database. We suggest that external consistency is desirable in most real–time transactions. Furthermore, external consistency may be more important than internal consistency for certain applications. Since real–time systems are to respond to external stimuli (e.g. in combat systems) or to control physical devices (e.g. in auto–pilot systems), a timely and externally consistent result is much more desirable than an out–of–date though internally consistent response. For instance, the trace of an unidentified object detected by an on–board system is externally consistent but may not be internally consistent before it is interpreted and filtered by the system. However, the system must avoid hitting the object before it is identified. Internal consistency is still desirable in most cases, but sometimes it can be maintained with a lower priority and may have a less stringent deadline.

### 2.2. Non–serializability of Real–Time Transactions

The decision of which consistency to enforce depends on the semantics of and the relationship between transactions. Although it is more complex, such an approach may provide better performance and should not be verified using the traditional serializability criteria. This is because real–time transactions in a system are usually *cooperative* rather than *competitive*. Different from the traditional database systems, transactions in a real–time system usually are well behaved and do not maliciously compete with each other for shared resources. For example, some real–time systems are implemented using a blackboard model in which any transaction may update any data using the current content of the blackboard [4]. Many transactions may be running at the same time and their executions are not serialized.

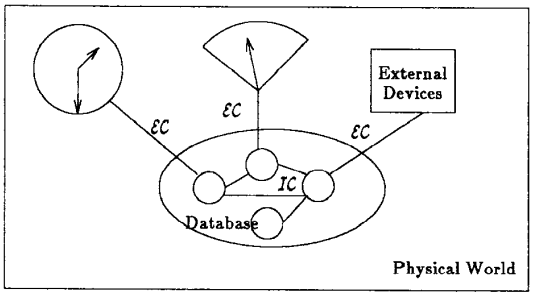
In real–time databases, external consistency may be more important than other factors for some applications. Indeed, a real–time database is to support transactions which interact with the real world. As long as the result of a transaction is consistent with the demands of the real world, whether the database is internally consistent or not is not even an issue. However, most real–time databases also include transactions whose correct execution depends on the databases' internal consistency. To facilitate these transactions, internal consistency is still desirable in most real–time databases. Depending on the semantics and requirements of operations, a real–time system may apply different measures at different time.

## 3. Models of Database Consistency

A database provides an environment in which transactions are processed. Databases are usually designed to embody all relevant physical world objects, so that they can facilitate user requests. Interactions with the physical world are conducted in form of transactions. The stored values in the database are determined from transaction executions which may or may not be the directly–entered information from the physical world. For certain systems, the database may be inconsistent with the real world for a short while when transactions are being processed but that is not a critical issue for many database systems. As long as all transactions are processed, the database will regain its consistency within the real world. Instead, the design of many databases emphasizes the maintenance of a consistent view from all components of the database, i.e. to avoid contradictory data records in the same database and to maintain what we call *internal consistency*. To guarantee internal consistency, database systems rely on the correct implementation of all transactions in the system and enforce the correct sequence of concurrent transaction executions by employing serializability theory [3].

### 3.1. External Consistency

In real–time (or embedded) systems, monitoring physical objects or controlling external devices is the main function of the systems. It is critical that the database mirror the *physical world.* The system is not self–contained but instead requires frequent interactions with the physical world to obtain and process timely information. In Figure 1, the relationship between a database and its physical environment is shown. The database is connected to some physical devices. Each device has a corresponding entity (shown as a circle) in the database. While the database may include some internal or *derived* information, it contains many data objects which have immediate counterparts in the physical world. It is important to keep the values of these objects in the database to be very close to their real values in the physical world. In this paper, such a property is referred to as *external consistency.*




*IC:* Internal Consistency
*EC:* External Consistency

**Figure 1.** Real–time system consistencies

Maintaining external consistency in the presence of deadlines may sometimes involve sacrificing internal consistency and serializability. For example, internal inconsistencies are sometimes resolved by undoing a transaction. Undoing external operations is usually impossible. To remove external inconsistencies, the only choice in many systems is to update the database so that it reflects the actual situation in the real world. For example, consider a system that is responsible for monitoring the activities of a robot. Suppose the robot performs an action that breaks an object into pieces. Once this action has taken place, it is impossible to undo the action. To regain external consistency in such a case, the database system can only enter the external incident in the database as promptly as possible. Then, to maintain internal consistency, the update is propagated to the rest of the database. If there are active transactions using the old invalid information, instead of enforcing serializability by blocking the update, those transactions in conflict may be interrupted or aborted. It is not as useful to continue transactions which are based on obsolete information.

Another example may illustrate the idea better. Suppose a transaction $T$ is to calculate the following function:

$$T(x,y) = f(x,y) + g(y) + h(x)$$

Assume the functions $f$, $g$, and $h$ are invoked in order, and the external value of $x$ changes between the invocations of $f$ and $h$. With the traditional serialization requirement, the update of $x$ must happen either before or after $T$ is executed. However, it may be more desirable for $T$ to use a timely data. If that is the case, instead of delaying the update until $T$ has completed, we may want to perform the update in the middle of $T$'s execution. The final result from $T$ thus becomes:

$$T(x,y,x') = f(x,y) + g(y) + h(x')$$

Sometimes the above function will produce a more acceptable (or up–to–date) result than the original $T(x,y)$, especially if $h$ is very sensitive to the change of $x$.

Sometimes a more externally consistent value may actually replace or even invalidate an old transaction still in execution. An example is a radar system which is to recognize approaching objects. When an object is distant, the signal is usually very fuzzy and very difficult to recognize. Sometimes, the recognition may take longer than the signal reading period. If a new signal arrives before an old recognition has been finished, it makes more sense to just abort the old recognition and start a new recognition using the closer picture which is more up–to–date.

### 3.2. Models of Transaction Schedules

In this section, we present some alternatives to serializable schedule. We define a real–time database to be a concurrent system with many transactions reading and writing objects in the database. A transaction is a sequence of distinct actions, $a_1, a_2, \cdots, a_n$. Each action can be a read or write on a certain object. For concurrent transactions, the following definition are adopted from [5].

**Definition.** A *schedule* $S$ of a set of transactions $\mathbf{T}$ is a sequence of actions, $a_i$, where

(1) for each $T \in \mathbf{T}$, the actions of $T$ either appear exactly once in $S$ or do not appear at all;

(2) if $a_i, a_j$ are actions of some $T \in \mathbf{T}$ and $a_i$ precedes $a_j$ in $T$, then $a_i$ precedes $a_j$ in $S$.

**Definition.** Two schedules are said to be *equivalent* if they have the same effect on the values of database and transaction results.

**Definition.** A *serializable* schedule is a schedule which is equivalent to a serial schedule (i.e. each transaction is executed sequentially).

Most database systems regard serializable schedules as the only class of acceptable schedules. This may be too restrictive for real-time systems. In [5], a class of acceptable but non-serializable schedules, called *semantically consistenct* schedule, is defined as follows.

**Definition.** A schedule $S$ is a *semantically consistent* schedule iff

(1) the execution of $S$ transforms the database into a consistent database;

(2) all sensitive transactions obtain a consistent view of the database.

A transaction is *sensitive* if it has external effects (i.e. the result is used outside of the database and is irrevocable), and it must be based on consistent databases. A semantically consistent schedule is acceptable since any external effect of the transactions is obtained from a consistent view of the database.

We adopt the concept of sensitive transactions in our model. In addition, we suggest that some transactions may be willing to accept a timely result from a database which is not totally consistent. One class of such transactions is called *operative* if the main objective of a sensitive transaction is to produce correct results based on a view of consistent with the external world. We thus define a new class of schedule as follows.

**Definition.** A schedule $O$ is *operationally consistent* iff

(1) the execution of $O$ transforms the database into an externally consistent database;

(2) all operative transactions produce timely results consistent with the state of the physical world.

In other words, an operationally consistent schedule preserves external consistency. Internal consistency may not be preserved as a result of $O$. If internal consistency is desired in the final database state after the execution of $O$, some compensating transaction may be conducted to regain the internal consistency. But as long as all external operations are consistent with the real world, an operationally consistent schedule may be acceptable and more responsive to the physical world.

### 3.3. Consistency Requirements of Transactions

Internal consistency of a database is defined by a set of constraints $\{c_i\}$ on data objects in the database. These constraints are usually well-defined when the database is created, and are used during database operations to maintain the valid relationship between objects in the database. As long as all $c_i$ are satisfied, the database is said to be internally consistent.

Conceptually, external consistency is concerned about the fact that an event has happened in the physical world but it may not has been entered in the database yet. Since not all real-world events will be recorded in the database, we are interested only in those related events which will be part of the database. We define a set of events, $E(t)$, which have occurred in the external world at time $t$. An event $e$ which has been recorded in database D is denoted by $e \subset$ D. Therefore, a database is said to be externally consistent at time $t$ iff $e_i \subset$ D, $\forall e_i \in E(t)$.

Databases are not always internally or externally consistent. Objects in a database can be in one of four different states defined by the combination of the two consistencies (Figure 2). An object can be internally consistent but not externally consistent (area I). This implies that its value may be out-of-date. An object can be externally consistent but not internally consistent (area II), if the value is up-to-date, but certain internal constraints are not satisfied. An object also can be both internally and externally consistent (area III), or neither (area IV).

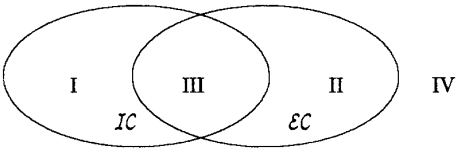



**Figure 2.** Database consistencies

Each transaction may require the database to be either internally and externally consistent to guarantee the correctness of its execution. In other words, only those data objects which satisfy the predefined consistency requirement can be used by the transaction. For example, a transaction printing a monthly bank statement requires an internally consistent view; while a transaction making the decision to shut down a power plant during an emergency may need a more externally consistent view.

Each transaction $T$ reads a set of data from the database to compute its result. These data are written by other transactions. Some of the transactions which may write these data are involved in maintaining the external consistency; we call the set of transactions to be the *depending* transactions of $T$ denoted by $dep(T)$. If $T$ is to provide an externally consistent response, those transactions in $dep(T)$ which are currently active must be executed before $T$. This is obvious since otherwise some data used by $T$ will be out-of-date. Whether or not other active transactions should be executed before $T$ depends on how close is $T$'s deadline and how long is the

execution time of each active transaction. In Section 4, we present a scheduling algorithm which allows transactions to be selectively executed depending on the requirements of internal and external consistency.

### 3.4. Classes of Real-Time Transactions

In many applications, it may be more efficient and acceptable to disregard serializability and to enforce external consistency. A transaction which handles timely information must decide if and when to ensure serializability. These decisions are often based on the semantics and the relations between individual transactions. It should be noted that we are not arguing that serializability is undesirable in databases. We simply suggest that, sometimes, to provide a timely response, serializability can be violated if the internal consistency can be recovered later or if it does not affect the transaction results. If meeting a deadline is more critical than producing a consistent result, the database must be able to facilitate that at the user's discretion. In this section, we discuss three different classes of transactions in terms of their internal consistency requirements.

(1) *Strong internal consistency*

Databases usually have more than one concurrent transactions at any time. Changes are made to the database as a result of transaction executions. Traditionally, no real-time constraint is required, and internal consistency is always guaranteed. As transactions are serviced, the interleaving of actions are controlled to maintain a consistent database state.

When a database is used in real-time systems, deadline constraints may be applied to transactions. If the time allowed for the completion of a transaction T is less than its required execution time, the database has two choices: either to abort T and undo all its effects on the database, or to allow T to produce an imprecise result [7,8]. To achieve a better performance, the database may want to adopt the latter approach if possible. However, with either approach, no transaction should be allowed to access an inconsistent part of the database. In other words, the internal consistency of a database is always maintained. For concurrent transactions $A$ and $B$ (with actions $A_i$ and $B_i$), the set of schedules acceptable to the database is the equivalent set of

$$A_1, A_2, A_3, A_4, B_1, B_2, B_3$$

This is the set of serializable schedule for A and B.

(2) *Weak internal consistency*

Not all actions in a transaction are involved in maintaining internal consistency. We can divide the actions in a real-time transaction into two parts: those actions in the *E−part* enter external events in the database (i.e. to maintain external consistency) and those in the *I−part* maintain internal consistency. Typically, a transaction may start with actions in the E-part and conclude with actions in the I-part.

In some real-time applications, the transaction cannot afford to wait for the database to regain the internal consistency if the transaction has a stringent deadline requirement. In such cases, the database may allow the internal consistency to be ignored temporarily in order to return a result before the transaction's deadline. However, in real-time systems, such transaction executions must still maintain the external consistency. One possible solution is to divide the transaction into two or more segments so that only the segment which maintains the external consistency is guaranteed to complete. The rest of the transaction may be executed at a lower priority after the result is produced before the deadline.

A similar situation occurs when a transaction B is blocked waiting on an active transaction A to finish. Although A has no deadline constraint, B must be finished before an approaching deadline. In this case, B can interrupt A as long as the E-part of A has finished. After B is executed, A may resume its I-part to recover the internal consistency. Thus for B's execution, internal consistency is not guaranteed, but external consistency is maintained. Since internal consistency usually has no time constraint, they may be regained after the external deadline is met. In this example, suppose $A_1$ and $A_2$ is the E-part of A, a possible and acceptable schedule may be as follows:

$$A_1, A_2, B_1, B_2, B_3, A_3, A_4$$

Such a schedule effectively divides A's deadline into two parts: the E-part still has a hard deadline, but the I-part has a soft deadline. In practice, the delayed I-part may be a little different from the original I-part since more work is involved to recover the internal consistency. We ignore such differences in this paper.

(3) *No internal consistency*

Some real-time transactions must always use timely information. In this class, maintaining external consistency for these transactions is the most important job of a system. One such example is the shuttle life-maintenance system which requires a fast reaction during an emergency. Regardless of the shuttle's original mission and other concurrent activities, the most important task in the situation is to keep track of the current human environment in the spacecraft, and to decide the best way to react.

A real-time database may require that consistency be ensured either externally or internally, with external consistency taking precedence over internal consistency. Thus, a transaction may either wait for an internally

consistent database before starts its execution, or, if it can guarantee that external consistency is guaranteed for the data it uses, proceed with the execution despite the concurrent activities in the database.

The reason why this is acceptable is because, in theory, real-time databases should reflect the real world. Ideally, the data used by a transaction should be always the same whether they are from the database or from the real-world. In reality, there usually is a delay between the occurrence of an external event and the recording of the event in the database. If a transaction uses externally consistent data, the transaction can be considered as serialized after the current state of the real world, which will be a consistent state of the database after all data are entered.

There are other situations where internal consistency is not a concern of the system. Certain protocols allow internal consistency to be maintained periodically. For example, deadlock detection in distributed systems is conducted once in a while or at the request of a process. Instrument reading processes do not have to follow a particular order in entering their data. Such situations demand a strong external consistency but weak or no internal consistency. A possible schedule for transaction A and B as defined above is:

$$A_1, A_2, B_1, B_2, B_3$$

After B's execution, either the internal consistency is already guaranteed by B, or it cannot be recovered by A's I-part. The I-part of A is useless in such cases and thus is removed from the schedule. Such a schedule may or may not guarantee the internal consistency in the final state, but external consistency is maintained so that the schedule is operationally consistent.

## 4. A Protocol for Ensuring Operational Consistency

With the requirement of operational consistency, transactions may be scheduled in situations when internal consistency cannot be maintained. In other words, concurrency control protocols which provide a serializable schedule may be too restrictive. In this section, we present a scheduling protocol which provides the desired flexibility. Before using the protocol, a compatibility table must be defined for all transaction pairs. During run-time, when real-time transactions are to be scheduled, the table is inspected to see if they are cooperative or competitive with earlier transactions. A transaction requiring an externally consistent data set must wait until all updates by its depending transactions are finished. We describe the compatibility table first.

### 4.1. Transaction Compatibility Table

In real-time databases, most real-time transactions are predefined and their computations well-known. By analyzing the relationship between transactions operations, a transaction compatibility table (TCT) can be created. A TCT (Figure 3) is similar to the locking table used in many concurrency control protocols. To decide whether a transaction T1 should be scheduled after T2, the entry of TCT(T1, T2) is examined. The entry has four possible values and consists of two relations. The first relation specifies the precedence of T1 with T2's E-part; the second the precedence of T1 with T2's I-part. The four values are as follows:

| | T1 | T2 | T3 |
|---|---|---|---|
| T1 | << | <- | <- |
| T2 | << | << | <- |
| T3 | <> | <> | << |

**Figure 3.** Transaction compatibility table (TCT)

<<:
This means T2 $\in$ *dep*(T1). T1's execution is dependent on T2's execution. So if T2 is waiting ahead of T1 in the scheduler queue, T2 must always be scheduled before T1.

<>:
This means T2 $\in$ *dep*(T1) but T2 contains an I-part which can delayed until later. T1 may preempt T2 when T2 reaches a externally consistent breakpoint (e.g. external data has been entered).

<-: This means T2 $\in$ *dep*(T1) but T2's I-part can be skipped if T1 has executed. Only the E-part of T2 must be finished before T1's execution. The I-part of T2 is not scheduled.

>>:
This means T2 is not in *dep*(T1). Thus it is acceptable to execute T1 before T2 even if T2 is waiting ahead of T1.

It should be clear that TCT is not symmetric. The entry of TCT(T1,T2) will be checked when T1 is waiting after T2 and vice versa. It is possible that T1<<T2 and T2<<T1, which simply means the two transactions are mutually dependent and their executions should be first-come-first-served. To demonstrate how the protocol works, we show some examples here. Suppose two real-time transaction T1 and T2 have the following actions:

T1: $R_1(y)\ W_1(y)$

T2: $W_2(x)\ W_2(y)$ **B** $R_2(y)\ W_2(z)$

Breakpoint **B** is defined after the E-part of T2. For each of the compatibility status, an acceptable schedule follows.

T1 << T2 : $W_2(x)W_2(y)R_2(y)W_2(z)R_1(y)W_1(y)$

T1 <> T2 : $W_2(x)W_2(y)R_1(y)W_1(y)R_2(y)W_2(z)$

T1 <- T2 : $W_2(x)W_2(y)R_1(y)W_1(y)$

In the first example, T1 is not allowed to interrupt T2 since it depends on the whole execution of T2. In the second example, T1 is run after T2's breakpoint. When T2 resumes, the newest value of $y$ is used in computing $z$ and internal consistency is maintained. In the last example all actions in the I–part of T2 are removed from the schedule, thus internal consistency is not guaranteed after the execution.

### 4.2. Scheduling Transactions Using TCT

Real–time databases usually have an on–line scheduler which makes decisions on which transactions to be executed next. Traditionally, the scheduler uses some simple timing conditions like deadline or slack to select one among all transactions which are ready to be executed. The problem of scheduling transactions with shared resources has been shown to be NP–complete [2,6]. Only sub–optimal schedules are feasible in most real–time databases.

Using the semantic information in a TCT, a scheduler may achieve a better performance by rearranging the transaction schedule. Suppose a transaction $T$ which has an early deadline is ready to be executed, and there are several other transactions waiting to be executed before $T$. The scheduler can compute the expected completion time for $T$ if all ready transactions are executed in the order of their arrival. If $T$ can meet its deadline in this schedule, no adjustment is required and $T$ will be placed at the end of the schedule.

If the expected completion time for $T$ is later than its deadline, the scheduler may adjust its starting time by using the TCT information. Suppose $S$ is in front of $T$ in the ready queue. If $S$ is not in $dep(T)$ and has a later deadline than $T$ (i.e. T>>S in TCT), the scheduler may move $S$ to be after $T$ in the ready queue. If $S$ is in $dep(T)$ but has an I–part which can be delayed (i.e. T<>S), $S$ is split into two sub–transactions: $S_e$ for the E–part and $S_i$ for the I–part. $S_i$ is then moved to after $T$. If $S$ is in $dep(T)$ but has an I–part which can be deleted (i.e. T<–S), $S_i$ is removed from the schedule. The process can be used by the scheduler repeatedly for transactions in front of $T$ until $T$ can be started early enough to finish before its deadline.

The algorithm apparently is not very efficient since to schedule a transaction, many transactions may have to be repositioned in the ready queue. We are investigating other protocols which may be more feasible for on–line scheduling. One novel approach is to fully analyze the possible scheduling bottleneck by an off–line scheduler. For each troubled queue pattern, a signature is created to identify it and an adjusted queue stored. During run–time, all it takes is to match the queue signature when a transaction may miss its deadline, and replace the ready queue with an adjusted one. The approach is possible if real–time transactions are mostly periodic so that their timing problems may appear over and over again in the same fashion. More research is still required to design efficient scheduling algorithms which can provide operationally consistent schedule in real–time databases.

## 5. Conclusions

It is generally agreed that real–time database systems have many open problems which require novel solutions [10]. What makes real–time databases so difficult to implement is the conflicting requirements of meeting deadlines and ensuring internal consistency. In many applications, it is unlikely that both requirements can be satisfied. Since a hard real–time system cannot change its deadline requirements, the only possible solution is to explore and modify the consistency requirement.

In this paper, we define a new correctness criteria for transaction schedules. We propose a new form of consistency which emphasizes the operational aspect of real–time databases. In real–time systems, responses to external events must be produced before their deadlines. As long as a database is consistent with the requirements of transactions, a timely and externally consistent performance is all that required to make a correct and useful real–time database.

## References


[1] ACM SIGMOD RECORD, *Special Issue on Real–Time Database Systems,* Vol. 17, No. 1, Mar. 1988.

[2] Blazewics, J., J.K. Lenstra and A.H.G. Rinnooy Kan, "Scheduling subject to resource constraints: Classification and complexity," *Disc. Applied Math.,* Vol. 5, pp. 11–24, 1983.

[3] Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman, *Concurrency and Recovery in Database Systems,* Addison–Wesley Publishing Company, Massachusetts, 1987.

[4] Dufree, E.H., et al., "Coherent cooperation among communicating problem solvers," *IEEE Transactions on Computers,* Vol. C–36, No. 11, pp. 1275–1291, Nov. 1987.

[5] Garcia–Molina, H., "Using semantic knowledge for transaction proccssing in a distributed database," *ACM Transactions on Database Systems,* Vol. 8, No. 2, pp. 186–213, June 1983.

[6] Garey, M.R., R.L. Graham, D.S. Johnson, and A. Yao, "Resource constrained scheduling as generalized bin packing," *J. Combinatorial Theory,* Vol. 21, pp. 257–298, 1976.

[7] Lin, K. J., S. Natarajan, and J. W. S. Liu, "Imprecise results: Utilizing partial computations in real–time systems," *Proc. Eighth Real–Time Systems Symposium,* San Jose, CA, Dec. 1987.

[8] Liu, J. W. S., K. J. Lin, and S. Natarajan, "Scheduling real–time periodic jobs using imprecise results," *Proc. Eighth Real–Time Systems Symposium,* San Jose, CA, Dec. 1987.

[9] Stankovic, J.A., "Real–time computing systems: The next generation," Technical Report, COINS 88–06, Univ. Mass., Amherst, MA, Jan. 1988.

[10] Watson, P., "An overview of architectural directions for real–time distributed systems," *Proc. 5th Workshop on Real–Time Software and Operating Systems,* Washington, DC, pp. 59–65, May 1988.