Real-Time Databases

Lisa Cingiser DiPippo and Victor Fay Wolfe

September 23, 1995

1 Introduction

Real-time databases manage time-constrained data and time-constrained transactions. They are useful in systems such as automated manufacturing, avionics, military command, control and communication, and programmed stock trading. In each of these applications, a computer system uses environmental data as input and must produce output to control its environment. Since data in the system represents the "current" state of the environment, the data is constrained to have been recorded recently enough to be considered valid. For instance, "current" sensor readings or "current" stock prices may be constrained to be no more than a few seconds old. Since environmental control must often be performed within a certain time interval to be correct, the transactions that operate on the data are also time-constrained. For instance, a military vehicle's evasive action, determined based on a transaction retrieving environmental data from the real-time database, may have to be performed by a certain deadline to be correct. This chapter describes the issues, research, and actual implementations of real-time database management systems.

A real-time database is a component of a *real-time system*. A real-time system is one in which timing constraints, such as start times, deadlines, and periods, must be met for the application to be correct. There has been a great deal of work in real-time systems that influence real-time database design. We summarize this work in Section 2 .

The presence of real-time requirements adds requirements to real-time database management. Typical database management systems enforce logical consistency constraints only, whereas real-time databases have *temporal consistency constraints*, such as validity intervals for data and time constraints for transaction execution, that must also be enforced. In Section 3, we describe the additional dimension of temporal consistency constraints and the requirements on the real-time database management system that it imposes.

To reason about, and express, these constraints, there have been several models proposed for real-time database systems. These models incorporate various aspects of traditional database models and various aspects of temporal consistency requirements. We describe two relational models and one object-oriented real-time database model in Section 4.

In the rest of the chapter, Section 5 describes two existing commercial real-time database management systems, one academic prototyping effort, and the Real-Time SQL standardization efforts. Section 6 summarizes current research in real-time database systems, focusing mostly on transaction scheduling and concurrency control. Section 7 summarizes what real-time database technology is available and indicates open questions in real-time database design.

2 Real-time Systems

In a real-time system, timing constraints must be met for the application to be correct. This requirement typically comes from the system interacting with the physical environment. The environment produces stimuli, which must be accepted by the real-time system within timing constraints. The environment further requires control output, which must be produced within timing constraints.

One of the main misconceptions about real-time computing is that it is equivalent to fast computing. Stankovic challenges this myth in [1] by arguing that computing speed is often measured in average case performance, whereas to guarantee timing behavior, in many real-time systems worst case performance should be used. That is, in a delicate application, such as a nuclear reactor or avionics control, where timing constraints *must* be met, worst case performance must be used when designing and analyzing the system. Thus, although speed is often a necessary component of a real-time system, it is often not sufficient. Instead:

predictably meeting timing constraints is sufficient in real-time system design.

This section describes the characteristics and requirements of real-time systems. It also describes work in real-time operating system scheduling, which is important when considering how real-time database temporal consistency can be enforced.

2.1 Real-time System Requirements

Real-time systems require that timing constraints be expressed, enforced, and their violations handled. The unit of time-constrained execution is called a *task*. In a real-time database,

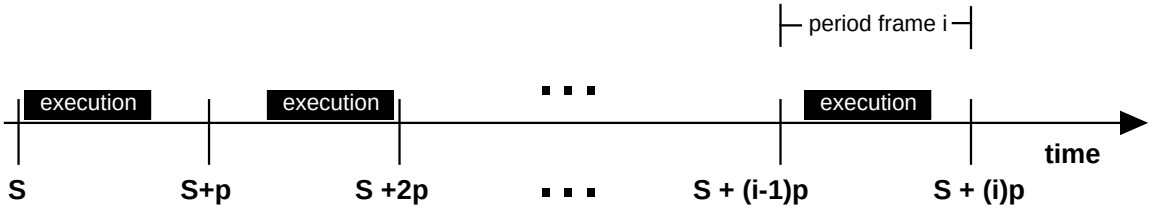

Figure 1: Periodic Execution

time-constrained transactions are considered tasks. Timing constraint expression can take the form of start times, deadlines, and periods for tasks. Timing constraint enforcement requires predictable bounds on task behavior. The handling of timing constraint violations depends on the tasks requirements: whether they are *hard, firm* or *soft* real-time. We now examine each of these aspects of timing constraints.

Expressing Timing Constraints. Most real-time systems specify a subset of the following constraints:

- An *earliest start time* constraint specifies an absolute time before which the task may not start. That is, the task must wait for the specified time before it may start.
- A *latest start time* constraint specifies an absolute time before which the task must start. That is, if the task has not started by the specified time, an error has occurred. Latest start times are useful to detect potential violations of planned schedules or eventual deadline violations before they actually occur.
- A *deadline* specifies an absolute time before which the task must complete.

Frequently, timing constraints will appear as *periodic execution constraints.* A periodic constraint specifies earliest start times and deadlines at regular time intervals for repeated instances of a task. Typically a *period frame* is established for each instance of the (repeated) task. As shown in Figure 1, period frame i specifies the default earliest start time and deadline for the i^{th} instance of the task. When periodic execution is originally started, the first frame is established, at time s in Figure 1. For periodic execution with period p, the i^{th} frame starts at time $s + (i-1)p$ and completes at time $s + (i)p$. As this indicates, the end of frame i is the beginning of frame $i + 1$. Each instance of a task may execute anywhere within its period frame.

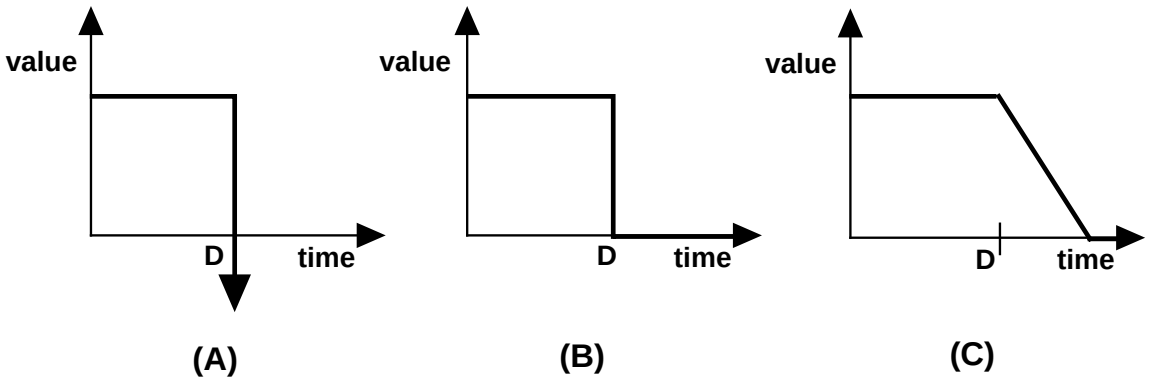

Figure 2: Modes of Real-Time

Modes of Real-time. Real-time constraints are classified as hard, firm, or soft, depending on the consequences of the constraint being violated.

A task with a *hard* real-time constraint has disastrous consequences if its constraint is violated. This characteristic is depicted in Figure 2A where the task causes a large negative value to the system if its deadline is missed. Many constraints in life-critical systems, such as nuclear reactor control and military vehicle control, are hard real-time constraints.

A task with a *firm* real-time constraint has no value to the system if its constraint is violated. This characteristic is depicted in Figure 2B where the task's value goes to zero after its deadline. Many financial applications have firm constraints with no value if a deadline is missed.

A task with a *soft* real-time constraint has decreasing, but usually non-negative, value to the system if its constraint is violated. This characteristic is depicted in Figure 2C where the task's value decreases after its deadline. For most applications, most tasks have soft real-time constraints. Graphic display updates are one of many examples of tasks with soft real-time constraints.

In some systems the mode of real-time is captured in a task's *importance* level. In systems such as Spring [2] task importance is categorized according to the mode of its timing constraint (hard, firm, soft). In other systems, importance is more general and tasks can be assigned importance relative to each other over a wider granularity of levels. Note that importance is not the same as *priority*. Priority, which is discussed in more detail in Section 2.2, is a relative value used to make scheduling decisions. Often priority is a function of importance, but also can depend on timing constraints, or some combination of these, or other task traits.

Predictability. In order to predictably meet timing constraints, it must be possible to accurately analyze timing behavior. To analyze timing behavior, the scheduling algorithm for each resource and the amount of time that tasks use each resource must be known. To fully guarantee this timing behavior, these resource utilizations should be worst case values; although some soft real-time systems can tolerate average case values that offer no strong guarantee.

Determining the resource usage time of tasks is often difficult. Results that can be obtained are often pessimistic worst cases with very low probability of occurring. Consider CPU utilization, which is one of the easier utilizations to determine. To establish a worst case CPU utilization, all conditional branches in the task must be assumed to take their worst path, and all loops and recursion must be assumed to have some bounded number of instances. Also, other task behavior, such as whether the task can be preempted from the CPU while waiting for other system resources, must be considered. For instance, if a task requires dynamic memory allocation, is it swapped off the CPU awaiting the memory allocation? CPU utilization is only one factor in a task's resource requirements. A task also needs resources such as main memory, disk accesses, network buffers, network bandwidth, I/O devices, etc. Furthermore, the use of these resources is inter-related and thus can not be computed in isolation. The problem worsens in database systems where "logical" resources such as locks on data structures and database buffers are additional resources to be considered. There has been work in determining worst case execution times [3], but in general this work makes limiting assumptions and/or produces very pessimistic results. Still, in order to guarantee, or at least analyze, the adherence to real-time constraints, resource utilizations of tasks must be known, so these rough estimates are used.

Assuming that worst case resource utilizations are known, analyzing timing behavior for predictability depends on the scheduling algorithms used. In the next subsection we discuss several real-time scheduling techniques and the forms of analysis that these techniques facilitate.

Imprecision. The introduction of timing constraints adds another dimension to real-time computing: it may need to be *imprecise*. That is, due to time "running out", the results produced by a task or set of tasks may not be exactly correct. In systems where timely but less accurate results are better than late exact results, imprecision may be tolerated. For instance, an air traffic control system may need a quick approximate position of an incoming aircraft rather than a late exact answer. Often the imprecision that is allowed must be within

a specified bound. For instance, the position data might have to be accurate within a few meters. More accurate data is desirable, but can be sacrificed if timing constraints do not permit it.

2.2 Scheduling

Real-time scheduling essentially maps to a *bin-packing problem* where tasks with known resource utilizations are the boxes, and the timing constraints establish the size of the bin. That is, each task can be considered a "box" whose size is its utilization of the resource being scheduled. The start times and deadlines of the tasks establish the boundaries of a "bin", or collection of bins, in which the boxes must be packed. The bin-packing problem is NP-hard, so optimal real-time scheduling, in general, is an NP-hard problem. However, hueristics have been developed that yield optimal schedules under some strong assumptions, or near-optimal results under less-restrictive assumptions.

There are many scheduling algorithms. Typically, a scheduling algorithm assigns *priorities* to tasks. The priority assignment establishes a partial ordering among tasks. Whenever a scheduling decision is made, the scheduler selects the task(s) with highest priority to use the resource. There are several characteristics that differentiate scheduling algorithms. They are:

- *Preemptive versus nonpreemptive.* If the algorithm is preemptive, the task currently using the resource can be replaced by another task (typically of higher priority).

- *Hard versus soft real-time.* To be useful in systems with hard real-time constraints, the real-time scheduling technique should allow analysis of the hard timing constraints to determine if the constraints will be predictably met. For firm and soft real-time, predictability is desirable, but often a scheduling technique that can demonstrate a best-effort, near-optimal performance, is acceptable.

- *Dynamic versus static.* In static scheduling algorithms, all tasks and their characteristics are known before scheduling decisions are made. Typically task priorities are assigned before run-time and are not changed. Dynamic scheduling algorithms allow tasks sets to change and usually allow for task priorities to change. Dynamic scheduling decisions are made at run-time.

- *Single versus multiple resources.* Single resource scheduling manages one resource in isolation. In many well-known scheduling algorithms, this resource is a single CPU.

Multiple resource scheduling algorithms recognize that most tasks need multiple resources and schedule several resources. *End-to-end* schedulers schedule all resources required by the tasks.

Rate Monotonic and Earliest-Deadline-First Scheduling. For a known set of independent, periodic tasks with known execution times, Liu and Layland proved that *rate-monotonic* CPU scheduling is optimal [4]. Here optimal means that if any scheduling algorithm can cause all of the tasks in a set to meet their deadlines, then rate-monotonic can too. Rate-monotonic scheduling is preemptive, static, single resource scheduling that can be used for hard real-time. Priority is assigned according to the rate at which a periodic task needs to execute: The higher the rate, which also means the shorter the period, the higher the priority. A supplemental result by Liu and Layland facilitates real-time analysis by proving that if the CPU utilization is less than approximately 69%, then the task set will always meet its deadlines [4]. For *dynamic* priority assignment that is also preemptive and single resource, Liu and Layland showed that earliest-deadline-first scheduling is optimal and that any task set using it with a utilization less than 100% will meet all deadlines.

Liu and Layland's elegant results come with strong assumptions. Among these assumptions is one which requires that tasks are independent. Subsequent work has relaxed many of their assumptions. Rajkumar, Sha and others have shown that task sets where the tasks can coordinate via mechanisms such as semaphores, can still be analyzed if they use *priority inheritance* protocols [5, 6]. In these protocols, a lower-priority task that blocks a higher-priority task (e.g. by holding a semaphore), inherits the priority of the higher-priority task during the blocking. With priority inheritance techniques, *priority inversion*, which is the time that a higher priority task is blocked by lower priority tasks, can be bounded and factored into the worst case execution time of each task. Utilization analysis, such as Liu and Layland's, can then be used to determine if timing constraints will be met [6]. Non-periodic tasks can also be accommodated using a *sporadic server* which periodically handles non-periodic tasks [7]. Real-Time Mach is a real-time operating system that employs many of these techniques [8].

POSIX Scheduling. The POSIX real-time operating system standards offer rudimentary real-time scheduling support in Unix-like systems. The POSIX standard mandates that the CPU scheduling be preemptive, priority-based, with a minimum of 32 priorities (see Figure 3). Individual implementations may offer more priorities, but the minimum is 32. The

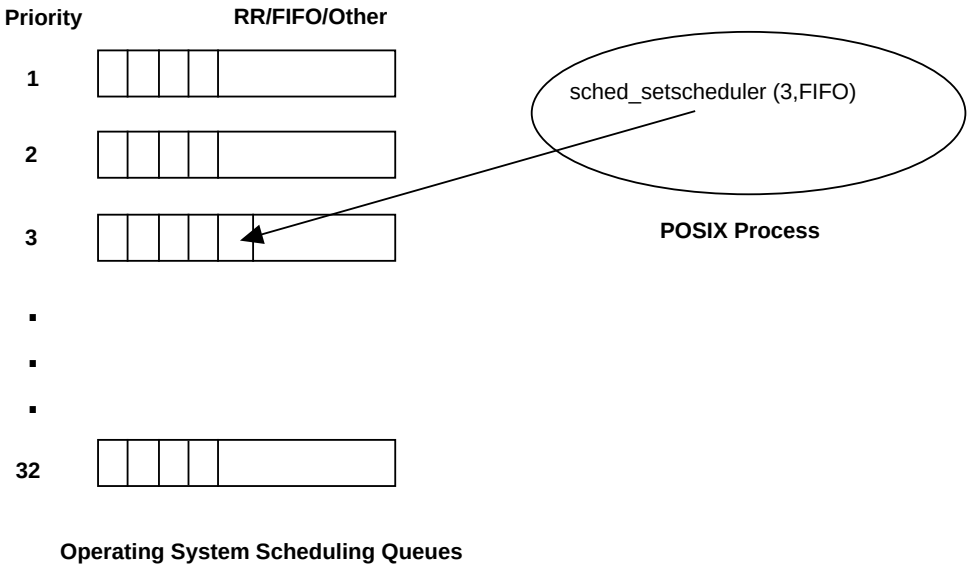

Figure 3: Real-Time Scheduling in POSIX Operating Systems

scheduling algorithm is simple: the highest priority ready task executes, possibly preempting a lower priority task. Tasks may dynamically change their own priority level or, in some cases, the priority level of other tasks. Within a priority level, tasks may be scheduled round-robin (with a system determined time quantum), or first-in-first-out (which is essentially round-robin with an infinite time quantum). This intra-priority scheduling is an unfortunate choice for real-time systems since it is not cognizant of timing constraints. Despite this limitation, rate-monotonic [9] and earliest-deadline-first [10] real-time schedulers have been built on real-time POSIX-compliant operating systems.

Resource Reservations. Zhao, Ramamritham, and Stankovic [11, 12] have developed real-time scheduling techniques that can handle dynamically arriving tasks that may use multiple resources. These techniques are based on resource reservations where each task attempts to reserve a time slot for itself during which it is guaranteed use of all resources that it requires. Tasks are allowed to make resource reservation requests based on a priority; the highest priority makes the request first. Priority is a weighted function of the task's deadline, execution time, and resource use. These techniques also allow for limited backtracking so that if allowing a task to make reservations before another task would cause deadlines to be missed, another pattern of reservations may be tried. They have demonstrated that their techniques achieve near-optimal results. Note that although these reservation techniques are not optimal, they have much less stringent assumptions than those of Liu and Layland. The

Spring kernel [2] is based on these scheduling techniques.

Scheduling Imprecise Computation. Liu, Lin et al [13] have proposed several algorithms for scheduling tasks that allow imprecise computation. In these algorithms, tasks are decomposed into a *mandatory* part and an *optional* part. The mandatory part is considered hard, the optional part is considered soft. Their algorithms attempt to schedule all mandatory parts of tasks and to schedule optional parts to minimize some error metric. The error metric indicates the consequences of not executing an optional part. [13] discusses several error metrics, each with a different scheduling algorithm that minimizes it. For instance, in systems with task importance levels, error might be weighted by each task's importance. The accompanying algorithm schedules optional parts of higher importance tasks whenever possible. Although this work is a good first step towards managing the imprecision in real-time systems, further work on guaranteeing that resulting imprecision is within system limits is needed.

3 Real-time Database Requirements

Real-time databases have all of the requirements of traditional databases, such as managing access to structured, shared, permanent data, but they can also require management of *time-constrained data* and *time-constrained transactions*. Furthermore, to facilitate analysis of timing behavior, certain real-time database functions may need to exhibit predictable timing behavior.

3.1 Temporal Consistency

Many requirements in a traditional database come from the desire to preserve the logical consistency of data and transactions. For instance, typical database management systems strive to maintain logical consistency by enforcing serializability of transactions and of operations on each data value. A real-time database additionally requires enforcement of *temporal consistency* constraints. That is, in a real-time database there are four forms of consistency constraints (summarized in Table 1): *Transaction logical consistency; data logical consistency; transaction temporal consistency;* and *data temporal consistency.*

Transaction Logical Consistency. Transaction logical consistency constrains the values of results produced by transactions. It is supported in most traditional databases manage-

	Temporal Consistency	Logical Consistency
Transaction	e.g. Start, Deadline, Period req	e.g. Serializability
Data	e.g. Absolute validity interval	e.g. Serializable operations

Table 1: Forms of Consistency for Real-Time Database

ment systems. For instance, *serializability* is a traditional transaction logical consistency correctness criteria. It requires that the results of transaction execution be equivalent to the results of some serial execution of the transactions. Techniques such as two-phase locking are designed to preserve serializability and therefore transaction logical consistency. In a real-time database, traditional transaction logical consistency may be "relaxed" to allow bounded imprecision, as discussed later in Section 3.2.

Data Logical Consistency. Data logical consistency is supported by most traditional database management systems. Range constraints, such as one constraining certain data values to be non-negative, are examples of data logical consistency constraints. Some consistency constraints are preserved by requiring that basic read and write operations on data be serializable. This can be done by typical read/write locking, for instance. Like transaction logical consistency, in a real-time database, data logical consistency may be "relaxed", as discussed later.

Transaction Temporal Consistency. Transaction temporal consistency constraints require that transactions be treated as real-time tasks (see Section 2) with timing constraints such as deadlines, start times, and periods. The constraints may be hard, firm, or soft. Violations of the timing constraints should be treated as a consistency violation by the database management system and appropriately recovered from.

Data Temporal Consistency. Data temporal consistency constrains how old a data item may be and still be considered valid. These constraints come from the fact that data used by time critical applications often must closely reflect the current state of the application environment. Data is collected at discreet intervals, and hence represents an approximation of reality. As time passes, this approximation becomes less accurate, until it reaches a point where the value is no longer reflective of the state of the environment. It is at this point in time that we say the data value is no longer temporally consistent. There are two forms of data temporal consistency constraints:

- *Absolute Temporal Consistency* - where a data item's age must be within a certain interval of the current time. For instance, a sensor value might have to have been recorded within two seconds of the current time to be considered absolutely temporally consistent.

- *Relative Temporal Consistency* - where several data values must have been recorded with the same time interval. For instance, if a sensor data value representing the speed of radar track and another representing the last measured position of the track are used to derive the new position, they may have to have been recorded within one second of each other for the derivation to be valid. This is a relative temporal consistency constraint between the speed and the last measured position data items.

3.2 Bounded Imprecision

As discussed in Section 2, real-time constraints may make precise computation impossible. In a database, a value is imprecise if it differs from the corresponding value resulting from each possible serializable schedule of the same transactions [14]. In many applications, some imprecision is tolerated. For instance, it may be sufficient for the position of a radar track to be within a few meters of its exact value. Although imprecision may be allowed, it must always be bounded. Thus, database logical consistency constraints for many real-time applications do not require the exact logical consistency that typical non-real-time databases do. Instead, the real-time database logical consistency constraints can allow bounded imprecision.

Imprecision can result from data management itself due to inherent conflicts between temporal and logical constraints. These conflicts make it difficult for a database management system to maintain all four forms of consistency constraints that are present in a real-time database (see Table 1 of Section 3.1). For example, in order to maintain precise transaction logical consistency (*e.g.* serializability), a transaction t_{update} that updates a piece of data x may be blocked by another transaction t_{read} that is reading x. If x is getting "old" it would be in the interest of its temporal consistency to allow t_{update} to execute. However, this execution could violate the precise data logical consistency of x or the precise transaction logical consistency of t_{read}. Thus, there is a trade-off between maintaining temporal consistency and maintaining logical consistency. If logical consistency is chosen, then there is the possibility that a piece of data may become old, or that a transaction may violate a timing constraint. If, on the other hand, temporal consistency is chosen, the consistency of the data or of the transactions involved may be compromised.

Priority inversion, mentioned in Section 2, is another example of the conflict between logical and temporal consistency. Priority inversion occurs because a lower-priority task/transaction is not preempted from a resource when a higher-priority task/transaction needs the resource. Liu and Layland's optimal scheduling results (which support temporal consistency) require such a preemption. Thus, priority inversion is the result of choosing precise logical consistency over a potential violation of temporal consistency.

A consequence of relaxing logical consistency, such as allowing non-serializable schedules or sequences of data operations, is that logical imprecision may accumulate in the data in the database and in the transactions' views of the data. Recall the example described earlier in which an update transaction t_{update} wishes to preempt a reading transaction t_{read}, in order to maintain the temporal consistency of the data. If this preemption is allowed, t_{read} may get an imprecise view of the data because it may read the value written by an uncommitted update transaction. The imprecision of a data item may be local to the view of a single transaction, such as when one transaction reads data written by another uncommitted transaction. A data item may also be imprecise with respect to future transactions that access it, such as when two transactions that write to the data item interleave.

Since imprecision may be inevitable in a real-time database, the database management system is required to manage the imprecision and bound it.

3.3 Predictability

As discussed in Section 2, predictable execution may be important in certain real-time applications. As a component of such systems, the real-time database may need to exhibit predictable behavior. This characteristic may generate several requirements for the database mangement system:

- Bounded worst case execution times of all database primitive commands.
- Bounded sizes of tables and data structures.
- Bounded use of memory.
- Bounded waits for database buffers.
- Bounded waits for secondary storage retrievals.
- Bounded blocking due to concurrency control.

- Bounded numbers and lengths of transaction aborts.
- Bounded indexing for locating data items.
- Use of real-time transaction scheduling that facilitates predictability.

As in all real-time applications, achieving such predictability often requires either making pessimistic worst case assumptions, using drastically simplified subsystems, or both.

3.4 Transactions

There are three types of transactions in a real-time database: *sensor* transactions, *update* transactions and *read-only* transactions [15]. All of these transactions can have temporal consistency requirements. Sensor transactions are write-only transactions that obtain the state of the environment and write the sensed data to the database. Sensor transactions are typically periodic. Update transactions can both read from and write to the database either periodically or aperiodically. Update transactions may be used to write values derived from computations or user input. Read-only transactions, such as some user queries, read data from the database and may also be either periodic or aperiodic.

Conventional transactions are structured to enforce the ACID properties [16]. These properties differentiate transactions from other tasks by facilitating reasoning about the logical consistency of transactions and data. However, the ACID properties ignore the temporal consistency requirements found in real-time databases.

To better support real-time applications, real-time databases redefine the ACID requirements to allow better support for temporal consistency while maintaining support for logical consistency. These definitions utilize semantic information to determine to what degree the ACID properties must be enforced. The redefinition of ACID properties for real-time transactions includes the following:

- *Atomic* - An atomic transaction implies all-or-nothing execution of the transaction. For real-time transactions, atomic execution is selectively applied to those pieces of the transaction that have a critical need for totally consistent data; instead of the transaction as a whole. Also, due to allowed logical imprecision, a "rollback" of a transaction in the "nothing" alternative of all-or-nothing may not rollback to the original state. Instead the transaction may be allowed to end in an inconsistent state as long as the resulting imprecision is bounded.

- *Consistent* - As discussed earlier, the consistency that typical ACID transactions seek to maintain is precise logical consistency. Real-time transactions must support a trade-off between temporal consistency and may logical consistency. This trade-off may introduce bounded temporal or logical imprecision.

- *Isolated* - Conventional transactions are required to have the property of appearing to have isolated execution. This implies that there be no dependencies in execution between transactions. In real-time databases, transactions may need to communicate and synchronize with other transactions to perform control functions. Transactions may need to synchronize on external time boundaries, system events, another transaction's results or end conditions, or they may need to perform some integrated set of tasks for an application that requires sharing of system state knowledge.

- *Durable* - The durability property of transactions implies that the results of a transaction are persistent and permanent. In a real-time database system, data must still be persistent, but not necessarily permanent. Temporal consistency constraints may indicate that some data is invalid and thus no longer needed. Another example is a circular queue data structure, which is commonly used in real-time databases to bound memory usage. This structure requires deletion of the oldest item when the queue is full and a new item is added. Durability of data must be specified semantically by the constraints and structure of the data, not by an implicit feature of a database or by a transaction's execution.

3.5 Summary of Real-Time Database Requirements

Most of the requirements discussed in this section are also specified in the Navy's Next Generation Computer Resources Database Standards Requirements Document [17] and in the specification for Real-Time SQL (see [18] and Section 5.4). To summarize, we list the requirements for real-time databases from [17] (which are assumed to be addition to the requirements of non-real-time databases):

1. *Modes of Real-Time* – A real-time database management system may support hard real-time, firm real-time, soft real-time, and/or non-real-time modes of operation.

2. *Real-time Transactions* – A real-time database management system requires the ability to allow users to issue real-time transactions where selected ACID properties are applied to parts of the transaction (note ACID properties are not required on an entire

transaction), and start events, deadlines, periods, and importance, of the real-time transactions are enforced.

3. *Data Temporal Consistency* – A real-time database management system requires the enforcement of absolute and relative data temporal consistency constraints.

4. *Real-time Scheduling* – A real-time database management system requires real-time transaction and operation scheduling for all resources allocated in the database system. The scheduling algorithm(s) should attempt to maximize meeting timing constraints and criticality (or some synthesis of these two attributes) of transactions, as well as attempting to maintain both logical and temporal consistency of data. Scheduling for hard real-time requires support for analysis of predictable timing behavior.

5. *Bounded Imprecision* – A real-time database management system may allow logical and temporal imprecision of data. It must provide the capability to constrain these imprecisions.

6. *Timing Constraint Violation Recovery* – A real-time database management system requires support for recovering from timing constraint violations of transactions and violations of temporal consistency of data.

7. *Predictability* – A real-time database management system is required to specify the probabilistic and worst case utilization amount and time for all resources (*e.g.* CPU time, memory, devices, data objects) of every DBMS function that can be used in hard real-time operation.

4 Real-Time Database Models

There have been several models developed to express the characteristics of real-time databases. We review two general models and one object-oriented real-time database model in this section.

4.1 Ramamritham Model

Ramamritham [15] presents a model of a real-time database in which both absolute and relative timing constraints are expressed on data. Transactions are characterized by the types and implications of their timing constraints.

Real-Time Data. A data object in the Ramamritham model is represented by *d:(value,avi,timestamp)*, where d_{value} represents the real-world data value, d_{avi} is the absolute validity interval of the data item and $d_{timestamp}$ is the time at which d_{avi} begins. Absolute temporal consistency of a data object d is maintained at a time t if $|t - d_{timestamp}| \leq d_{avi}$.

A set of data objects used to derive another data object is stored in a relative consistency set, R. R_{rvi} is the relative validity interval of the set. Relative temporal consistency is maintained as long as the timestamps of each data object in R are within R_{avi} of each other. In other words, $\forall_{d,d' \in R} |d_{timestamp} - d'_{timestamp}| \leq R_{rvi}$. The timestamp of derived data is a function of the timestamps of the data used to derive it.

Consider the following example: Let $temperature_{avi} = 5$, $pressure_{avi} = 10$, $R = \{temperature, pressure\}$ and $R_{rvi} = 2$. If $current\_time = 100$, then $temperature = (347, 5, 95)$ and $pressure = (50, 10, 97)$ are temporally consistent. However, $temperature = (347, 5, 95)$ and $pressure = (50, 10, 92)$ are not temporally consistent because even though the absolute consistency requirements are met, R's relative consistency is violated.

Real-Time Transactions. Transactions in the Ramamritham model are characterized along three dimensions: the way in which the data is used by a transaction, the nature of the timing constraints, and the implication of a missed deadline.

Real-time transactions may use data in one of three ways. *Write-only transactions* write to the database. Sensor transactions are generally write-only. *Update transactions* derive data, by reading and performing calculations, and store it in the database. And *read-only transactions* read data from the database.

Timing constraints on transactions come from temporal consistency requirements of the data or from requirements imposed on system reaction time. For instance, an update transaction may be required to execute every 5 seconds because the data that it writes has an absolute validity interval of 5 seconds. On the other hand, another transaction may need to be performed within a certain amount of time to satisfy external requirements. For example, the constraint:

If $temperature > 1000$,
then *within 10 secs* add coolant to reactor

requires that the transaction to add coolant be executed within 10 seconds. The effect of missing a deadline is the third way of characterizing transactions. The model uses *hard, firm* and *soft*, as described last section, as values of this characteristic.

4.2 Kim/Son Model

In [19] Kim and Son present a model of real-time databases that draws some of its concepts from the Ramamritham model of [15]. The model has been broadened to include non-real-time data and to classify transactions based upon the type of application in which they may be used.

Real-Time Data Objects. The Kim/Son model divides data objects into two types: *continuous* and *discrete*. Continuous objects are objects whose value can become invalid with time. Such objects can be obtained directly from a sensor (*image object*) or computed from the values of other objects (*derived object*). Discrete data objects are non-real-time objects in that their values do not become obsolete with time.

Each continuous data object has associated with it a *timestamp* which tells when the current value of the data object was obtained. The *absolute validity duration* of a continuous data object is the length of time during which the value of the object is considered valid. A *relative validity duration* is associated with a set of data objects Σ_y used to derive a new data object y. A set Σ_y is relatively temporally consistent if the *temporal distance* between y and any data object in Σ_y is not greater than the relative validity duration rvd_y.

Real-Time Transactions. Real-time transactions in the Kim/Son model are characterized by: the implication of missing a deadline (hard, critical (firm), soft real-time); arrival pattern (periodic, sporadic or aperiodic); data access pattern (write-only, read-only, update, or random); data requirement (known or unknown); run-time requirement (known or unknown); and accessed data type (continuous, discrete or both).

Given the types of data objects described above and the characterization of real-time transactions, there are hundreds of possible transaction classes. However, only some of these classes are feasible in a real-time database. For instance, it does not make sense to have a hard real-time transaction with random arrival pattern, random data access set, and unknown execution time. The Kim/Son model classifies transactions based upon how an application may use them. Table 2 summarizes the transactions classes.

Class I transactions are hard real-time periodic transactions. Such transactions have all data and run-time requirements available in advance. They represent the only writing source for continuous data objects, and thus it is feasible to guarantee their hard timing constraints. This class of transactions can be further broken down into subclasses.

Class IA transactions maintain the temporal consistency of continuous data objects.

They are write-only and since each such transaction is the only writer to a continuous data object, there are no conflicts between class IA transactions. Transactions of class IB are update transactions. They read some data objects, compute new values and write to derived data objects. They do not conflict with other class I transactions because each data object has only one writer. Class IC transaction periodically retrieve data values from the database. They are read-only transactions with hard deadlines.

Class II transactions are read-only transactions with some critical timing constraints. These timing constraints come from system response time requirements and not data temporal consistency requirements. Because class II transactions can access both discrete data objects (which require serializable access) and continuous data objects, timing constraints cannot always be met. However, since the only source of unpredictability in class II transactions is the data requirement, the transaction execution time is a function of only one variable.

All transactions not belonging to any of the other classes can be categorized as class III transactions. They have either soft or firm deadlines, their data and run-time requirements are not always known, and they access both continuous and discrete data objects.

Most real-time applications have transactions in each of the above classes. Consider a medical information system for example. Class IA transaction are those that update the dynamic physical status of a critical patient from the sensor devices, such as blood pressure, heart rate and body temperature. Transactions that write derived information from the raw data are class IB transactions. Transactions monitoring the physical status of the patient can be categorized as class IC transactions. A class II transaction in this example might be a decision-making transaction during a critical operation on a patient. Such a transaction may access not only the patient's current physical status, but also his or her medical history. Class III transactions in this example include record-keeping transactions such as retrieving weight or height.

4.3 RTSORAC Model

The RTSORAC model [20] incorporates features that support the requirements of a real-time database into an extended object-oriented model. It has three components that model the properties of a real-time object-oriented database: *objects*, *relationships* and *transactions*.

Objects. Objects represent database entities. Each *object* consists of five components, $\langle N, A, M, C, CF \rangle$, where N is a unique name or identifier, A is a set of attributes, M is

	Class I			Class II	Class III
Property	A	B	C		
Timing constraints	Hard			Critical	Soft or firm
Arrival pattern	Periodic			Sporadic	Aperiodic
Data access pattern	Write-only	Update	Read-only	Read-only	No restriction
Data requirement	Known			Unknown	Unknown
Run-time requirement	Known			Unknown	Unknown
Update data type	Image	Derived	N/A	N/A	Discrete
Correctness criteria	Temporal consistency			Both	Logical consistency
Transaction schedule	Non-serializable			Both	Serializable
Performance goal	100% guarantee			Statistical	No guarantee, but best-effort

Table 2: Classification of Kim/Son Real-Time Transactions

a set of methods, C is a set of constraints, and CF is a compatibility function. Figure 4 illustrates an example of a **Train** object (adapted from [21]) for storing information about a train control system in a database.

Each attribute of a RTSORAC object is characterized by $\langle N_a, V, T, I \rangle$. N_a is the name of the attribute. The V field is used to store the value of the attribute, and may be of some abstract data type. The T field is used to store the time at which the value was recorded. The I field of an attribute is used to store the amount of imprecision associated with the attribute, and is of the same type as the value field V.

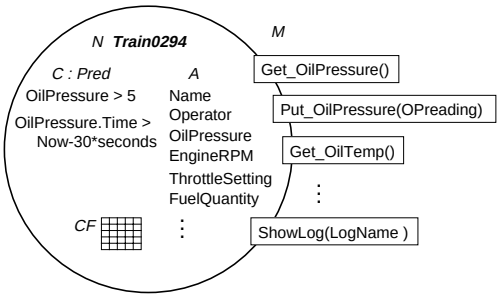

Figure 4: Example of **Train** object

Each method of an object is of the form $\langle N_m, Arg, Exc, Op, OC \rangle$. N_m is the name of the method. Arg is a set of arguments for the method, where each argument has the same components as an attribute, and is used to pass information in and/or out of the method. Exc is a set of exceptions that may be raised by the method to signal that the method has terminated abnormally. Op is a set of operations that represent the actions of the method. These operations include statements for conditional branching, looping, I/O, and reads and writes to an attribute's value, time, and imprecision fields.

The OC characteristic of a method is a set of operation constraints. An operation constraint is of the form $\langle N_{oc}, OpSet, Pred, ER \rangle$ where N_{oc} is the name of the operation constraint, $OpSet$ is a subset of the operations in Op, $Pred$ is a predicate (Boolean expression), and ER is an enforcement rule. The predicate is specified over $OpSet$ to express precedence constraints, execution constraints, and timing constraints [22]. The enforcement rule is used to express the action to take if the predicate evaluates to false. A more complete description of an enforcement rule can be found in the paragraphs below describing constraints.

Here is an example of an operation constraint predicate in the **Train** object:

$Pred$: `complete(Put_OilPressure) < NOW + 5*seconds`

A deadline of `NOW + 5*seconds` is specified for the completion of the `Put_OilPressure`

method.

The constraints of an object permit the specification of correct object state. Each constraint is of the form $\langle N_c, AttrSet, Pred, ER \rangle$. N_c is the name of the constraint. $AttrSet$ is a subset of attributes of the object. $Pred$ is a predicate that is specified using attributes from the $AttrSet$. The predicate can be used to express the logical consistency requirements by using value fields of the attributes. It can express temporal consistency requirements by using the time fields of attributes. It can express imprecision limits by using the imprecision fields of attributes.

The enforcement rule (ER) is executed when the predicate evaluates to false, and is of the form $\langle Exc, Op, OC \rangle$. Exc is a set of exceptions that the enforcement rule may signal, Op is a set of operations that represent the actions of the enforcement rule, and OC is a set of operation constraints on the execution of the enforcement rule.

As an example of a temporal consistency constraint, consider the following. As mentioned earlier, the **Train** object has an oil pressure attribute that is updated with the latest sensor reading every thirty seconds. To maintain the temporal consistency of this attribute, the following constraint is defined:

```
N :        OilPressure_avi
AttrSet :  {OilPressure}
Pred :     OilPressure.time > Now - 30*seconds
ER :       if Missed <= 2 then
             OilPressure.time = Now
             Missed = Missed + 1
             signal OilPressure_Warning
           else signal OilPressure_Alert
```

The enforcement rule specifies that if only one or two of the readings have been missed, a counter is incremented indicating that a reading has been missed and a warning is signaled using the exception `OilPressure_Warning`. If more than two readings have been missed, then an exception `OilPressure_Alert` is signaled, which might lead to a message being sent to the train operator. The counter `Missed` is reset to zero whenever a new sensor reading is written to the `OilPressure` attribute.

The compatibility function of an object expresses the semantics of allowable concurrent execution of each ordered pair of methods in the object. For each ordered pair of methods, (m_i, m_j), a Boolean expression ($BE_{i,j}$) is defined. $BE_{i,j}$ can be evaluated to determine whether or not m_i and m_j can execute concurrently. Based on the semantics of the application, the compatibility function may allow method interleavings that introduce imprecision into the attributes and method arguments. Therefore, in addition to specifying compat-

	Compatibility	Imprecision Accumulation
A:	$CF(Put\_OilPressure(), Put\_OilTemp()) = TRUE$	No Imprecision
B:	$CF(Get\_OilPressure(P_1), Put\_OilPressure(P_2)) = (OilPressure.time <= Now - 30 * seconds)\ AND\ (\|OilPressure.value - P_2.value\| <= (P_1.implimit - P_1.ImpAmt)$	Increment $P_1.ImpAmt$ by $\|OilPressure.value - P_2.value\|$

Figure 5: Compatibility Function Examples

ibility between two method invocations, the compatibility function expresses information about the potential imprecision that could be introduced by interleaving method invocations. There are three potential sources of imprecision when methods invocations m_i and m_j are interleaved: imprecision in the value of an attribute that is written by both m_i and m_j, imprecision in the value of the return arguments of m_i when m_i reads attributes written by m_j and imprecision in the value of the return arguments of m_j when m_j reads attributes written by m_i [14].

Figure 5 demonstrates several examples of the compatibility function and its associated imprecision accumulation for the **Train** object of Figure 4. In Example **A** of Figure 5, the compatibility function is used to specify that the methods `Put_OilPressure` and `Put_OilTemp` can always run concurrently. This is appropriate because these two methods access different attributes. No imprecision is introduced in this case. Example **B** demonstrates trading off logical consistency for temporal consistency. If the temporal consistency constraint on the `OilPressure` attribute has been violated ($OilPressure.time <= Now - 30 * seconds$), then the compatibility function specifies that the `Put_OilPressure` method invocation can execute concurrently with an active `Get_OilPressure` method, presumably to restore the temporal consistency of the `OilPressure` attribute. The CF restricts this interleaving to occur only if the amount of imprecision in the argument P_1 of the `Get_OilPressure` method invocation does not exceed the limit specified by the invoking transaction ($P_1.implimit$). The amount of imprecision to add to P_1 in this case is also specified by the compatibility function. Note that although we use only simple methods (essentially reads and writes) in this example, the compatibility function can specify imprecision accumulation for general object methods [23].

Note how the RTSORAC model supports the trade-off of logical consistency for temporal consistency found in real-time databases. Object designers can semantically express their preferences toward logical or temporal consistency in certain situations by using the RTSORAC compatibility functions. The imprecision field of each attribute allows accumulation of the imprecision that could result from the trade-off. The model can use constraints on imprecision fields to express limits on the allowed amount of imprecision.

Relationships. Each relationship in the RTSORAC model represents an aggregation of two or more objects, and consists of $\langle N, A, M, C, CF, P, IC \rangle$. The first five components of a relationship are identical to the same components in the definition of an object. In addition, objects that can participate in the relationship are specified in the participant set P, and a set of interobject constraints is specified in IC.

Figure 6 illustrates an example of an **Energy Management** relationship for relating a **Train** object with a **Track** object. The **Track** object stores information such as track profile and grade, speed limits, maximum load, and power available. The **Energy Management** relationship uses both train and track information to determine control algorithm parameters such as fuel efficient throttle and brake settings.

Each participant in a relationship is of the form $\langle N_p, OT, Card \rangle$. N_p is the name of the participant. OT is the type of the object participating in the relationship. $Card$ is the cardinality of the participant, which is either *single* or *multi* [24]. Constraints can be used to express cardinality requirements of the relationship, such as minimum and maximum cardinality of the participants. In Figure 6, `Train` and `Track` are single cardinality participants.

The interobject constraints placed on objects in the participant set are of the form: $\langle N_{ic}, PartSet, Pred, ER \rangle$. N_{ic}, $Pred$, and ER are as in object constraints, and $PartSet$ is a subset of the relationship's participant set P. The predicate is expressed using objects from the $PartSet$, allowing the constraint to be specified over multiple objects participating in the relationship. Enforcement rules are defined as before by $\langle Exc, Op, OC \rangle$, however the operations in Op can now include invocations of methods of the objects participating in the relationship.

As an example of an interobject constraint, consider the **Energy Management** relationship in Figure 6. A **Train** object will be on one specific segment of track, represented by the **Track** object participating in the relationship. The train should obey the speed limits set on the track segment, so the following interobject constraint predicate could be specified:

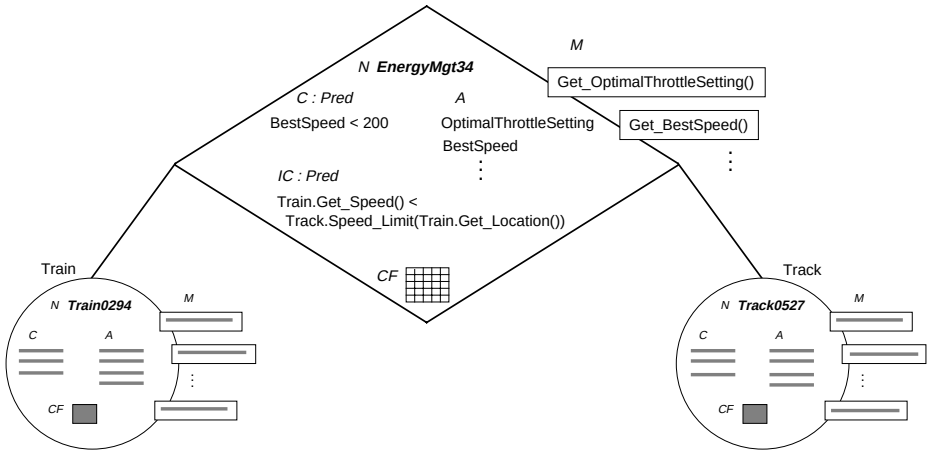

Figure 6: Example of **Energy Management** relationship

$Pred:$ `Train.Get_Speed() < Track.Speed_Limit(Train.Get_Location())`

If the speed of the train exceeds the speed limit posted at the train's location on the track, then the corresponding enforcement rule signals `SpeedLimitExceeded`.

Transactions. A RTSORAC *transaction* has six components, $\langle N_t, O, OC, PreCond, PostCond, Result \rangle$, where N_t is a unique name or identifier, O is a set of operations, OC is a set of operation constraints, $PreCond$ is a precondition, $PostCond$ is a postcondition, and $Result$ is the result of the transaction. Each of these components is briefly described below.

The operations in O represent the actions of the transaction. They include statements of the language in which the transaction is written, and method invocations on database objects (MI). Method invocations (MI) are of the form $\langle MN, ArgInfo \rangle$, where MN is the method name (prepended with the appropriate object identifier) and $ArgInfo$ is a set of tuples containing argument information. Each argument tuple is of the form $\langle aa, maximp, tcr \rangle$ where aa is the actual argument to the method, $maximp$ is the maximum allowable imprecision of the argument, and tcr is the temporal consistency requirement of the argument. The fields $maximp$ and tcr are specified only for arguments that are used to return information to the transaction. These fields allow the transaction to specify requirements that differ from those defined on the data in the objects. For example, the transaction might be willing to accept a value whose temporal consistency requirements have been violated so as to meet other timing constraints. The data may still be useful to the transaction because of other available information (for example, it may be able to do some extrapolation). A transaction

may also specify that data returned by a method invocation must be precise (*maximp* is zero).

OC is a set of constraints on operations of the transaction. These constraints are of the same form as the operation constraints specified for methods, $\langle N_c, OpSet, Pred, ER \rangle$. They can be used to express precedence constraints, execution constraints, and timing constraints. For example, a transaction may require that a sensor reading be returned within two seconds.

PreCond represents preconditions that must be satisfied before a transaction is made ready for execution. For example, it may be appropriate for a transaction to execute only if some specified event has occurred. The event may be the successful termination of another transaction, or a given clock time. *PostCond* represents postconditions that must be satisfied upon completion of the operations of the transaction. The postconditions can be used to specify the semantics of what constitutes a *commit* of a transaction containing subtransactions. *Result* represents information that is returned by the transaction. This may include values read from objects as well as values computed by the transaction.

5 Real-Time Database Systems and Standards

This section reviews several real-time database development efforts. It first examines the only two commercial products to advertise themselves as "real-time databases": Zip RTDBMS for DBx, and EagleSpeed RTDBMS from Martin Marietta. It also presents an academic prototype real-time oject-oriented database based on Texas Instrument's Open Object-Oriented Database system. Finally, it reviews the proposed Real-Time SQL standard. These reviews are done using the requirements for real-time databases that were presented in Section 3.

5.1 The Zip Real-time Database System

ZIP RTDBMS from DBx, Inc. is proclaimed to be a memory-resident, high-speed, real-time database management system. Version 1.2 currently runs on LynxOS 2.2 operating system, which is consistent with the POSIX 1003.4 real-time operating systems standards. Implementations of Zip for other platforms are also under development. Zip provides bounded response time, static definition and evaluation of schemata and data access behavior, and *a priori* query optimization before run-time. It features preallocation of system resources at database creation time as well as fixed-time attribute and index resolution at run-time.

The bounded response time is designed for atomic operations such as insertion, deletion, etc., performed directly on the data. Some response times collected from Zip average-case

Function	Timing
`createDB`	1 sec
`connectDB`	90 ms
`insertDB`	520 us
`deleteDB`	820 us
`destroyDB`	40 ms
`disconnectDB`	50 ms

Table 3: Zip RTDBMS Average-Case Timing under LynxOS 2.2 i486/50

timing sweep are shown in the Table 3.

In Zip, all tables and queries are defined statically in a user-created data definition language (DDL) file. The DDL has some similarities to standard language such as SQL, but it is Zip-specific. The DDL file is parsed by a Zip utility and converted into a schema binary file, used by both the client and the database server. The schema file contains database specifications, table, and index definitions. An example of a data definition file, simplified for demonstration is presented in Figure 7.

In the example, the database specifications are defined in lines 1-5, along with one table in lines 6-13, and an index at 15-18. The table contains a sequence of variables (columns) of various data types. Zip supports a total of 12 types plus the string data type. One distinguished variable in the table is `timestamp`, which is not implemented, but proposed for the next release of Zip. The timestamp supports the determination of temporal consistency of data.

After a table has been declared in Zip's DDL, it must now be set to a particular type and size. In the example of Figure 7 the table type is `STATIC` and its size is `10000`. Three types of data relation (table types) are supported by Zip. `STATIC` relations deal with records of fixed size and a relatively stable content. Only `SELECT` and `IPDATE` operations can be used with these tables. `BOUNDED` relations are of fixed maximum size, but with time-varying content. All operations are allowed on these tables. `ROLLING` relations resemble a fixed size queue where the new insertions force the old data down the queue until eventually there is no more "room" for it and it is lost, hence simulating "aging" of data. The last type is important in many real-time environments where the value of the data changes with time and fixed-sized resources are necessary.

The last statement in the example DDL file is a declaration of the table index. It identifies the *key variable*, the *load factor*, which is the percent of free spaces, and the number of *distinct keys* which are used to compute optimal hashing methods for the table. Although though

```
1: define database db_name ("sampleZipDB");
2: define database db_dir_path ("/local/db");
3: define database db_phys_addr (0x0);
4: define database db_size (393216);
5: define database db_grants (0777);

6: create table calibrate_parameters (
7: unit_id  byte2 not null, unique
8: cal_value byte8f not null,
9: tolerance byte8f,
10: descript string[4],
11: cal_cycle byte2,
12: cal_date date not null,
13: timestamp tstamp );

14: define table calibrate_parameters (STATIC, 10000);
15: define index unit_id_idx on calibrate_parameters (unit_id)
16: ideal,
17: load (100%),
18: distinct (10000);
```

Figure 7: Example Zip DDL

```
1: declareDBrelation(DBhandle, "stock")
2: bindDBvariable(DBhandle, id_name, &id, &id_type)
3: bindDBvariable(DBhandle, price_name,&price,&price_type)
4: tids = selectDB(DBhanle,DEFAULT_FUNC)
```

Figure 8: Example of Zip Query

).

any query could be generated, only the queries created with the indices in mind are efficient.

Since the schema file is available before the creation of the database, most of the system allocation is done during the creation. Once running, the server can be accessed by multiple clients via a multitude of connections, each according to the specifications defined in the schema file. It is worth noting that the version of Zip we review here does not have any user-friendly interface for generation of the DDL files, but DBx claims plans for SQL compatibility in the future versions. Along with all other components of the DDL, file queries must be created by hand in the client program itself (see Figure 8).

The example of Figure 8 shows a simple query that selects all tuples from a table called `stock`, containing two columns: `id` and `price`. The table is assumed to have been declared earlier. The `bindDBvariable` function specifies to the database what columns to work with while processing a query. The function's parameters contain a database handle, column name and type information. In the example query the columns in lines 2 and 3 are bound and then a selection is performed in line 4 with the `selectDB` function. This function accepts the database handle and error-function as parameters. The return value `tids` is a pointer to an array of tuples that was selected. Once selected, these tuples can be traversed with the `nextDBtuple` function (not shown) and accessed individually.

The Zip RTDBMS claims quick execution times because the system is relieved of much of the traditional database functionality, such as query parsing, relation and attribute resolution, and index selection. In addition, the whole database resides in a shared memory special file with every access calculated as a base plus an offset pair. The implementation is multi-threaded and optimized for the underlying real-time operating system environments.

Though features described above pertain in part to those found in a real-time database, Zip RTDBMS clearly lacks some of the major components. This is because Zip was designed for hard real-time, which requires the elimination of many features in order to preserve predictability. Most of the characteristics of real-time transactions mentioned in Section 3 are

not supported, nor is dynamic real-time scheduling. Data temporal consistency, both absolute and relative are not supported, however functions such as `get_earliest_timestamp`, `get_latest_timestamp` and `testDBtimestamp` are proposed and thus would be available for client applications to implement. Zip does not perform any logical or temporal imprecision data management, hence bounded imprecision support cannot be claimed. Time constraint violation recovery is non-existent, since neither transactions nor data temporal consistency violations can be detected. Concurrency control for clashing requests is not real-time, instead Zip chooses standard locking in all cases. Predictability is one issue that Zip supports well by providing bounded response time on all database calls, though at the sacrifice of unlimited storage capability. Overall, Zip RTDBMS is perhaps better considered a real-time data store, rather than a real-time database. It claims only one of the real-time database features of Section 3.

5.2 The EagleSpeed Real-Time Database System

The EagleSpeed real-time database management system is the commercial release of a database designed for a submarine command system. It is based on the ANSI CODASYL (network) data model. It is designed to support hard real-time applications. The stated goals of EagleSpeed are:

- To synchronizing with environmental processes that must be controlled.
- To support *a priori* determination of system schedulability.
- To provide predictability and punctuality of transaction access.
- To provide speed, determinism, and minimalism.

It has achieved the predictable execution time required by hard real-time in several ways. First, EagleSpeed implements only a subset of database management functionality. The removal of database system functionality places more burden on the application writer. It only provides a subset of the CODASYL verbs; requiring the applications writer to define other functionality. Second, by utilizing the network data model's ability to fix logical data structures in advance, it fixes access time to the data. Third, it uses one single-layer schema, which mimizes overhead due to mapping levels and due to maintaining metadata. Fourth, the physical location of data objects can be fixed to facilitate retrieval.

Transactions are Ada programs with ACID properties. They are invoked via messages sent to their site of storage. In addition, the system allows direct access to the transaction

manager and the operating system to facilitate real-time performance. Timing constraints on transactions are not supported since it assumed that *a priori* analysis will determine if all transactions meet timing constraints. Transactions are given priorities and scheduled by the underlying operating system. Concurrency control is handled by strict two-phase locking with no imprecision allowed. Like Zip, EagleSpeed is a fast, predictable real-time data store, but lacks some features of a full real-time database.

5.3 Real-Time Extensions to the Open Object-Oriented Database System

A prototype object-oriented real-time database system has been designed at the University of Rhode Island [25]. It implements the RTSORAC model described in Section 4.3, by extending the Open Object-Oriented Database System (Open OODB) [26]. The Open OODB system was initiated by the U.S. Advanced Research Projects Agency (ARPA). The original goal of Open OODB is to establish a common, modular, modifiable, object-oriented database system suitable to be used by a wide range of researchers and developers [26]. Open OODB is designed so that features such as transaction management, query interface, persistence, etc. are modules that can be individually "unplugged" and replaced by other modules.

The basic conceptual system architecture of Open OODB is shown in Figure 9 (along with the real-time extensions). The *support managers* are modules that are currently implemented as library routines that get linked into the user's C++ program to (transparently) provide the extended database capabilities. *Policy managers* (PMs) provide extenders to the behavior of programs by coordinating the support managers. For instance, the *Persistence Policy Manager* provides applications with an interface through which they can create, access, and manipulate persistent objects in various address spaces. The *Transaction Policy Manager* enables concurrent access to persistent and transient data; its implementation in the current alpha release is a trivial mapping to Exodus write locks on all objects. Other policy managers include those for distribution, change management, indexing, and query processing. The query interface is in two forms: an extended version of C++ and an SQL-like language called OQL, which must be embedded in C++ code [26].

Real-time Extensions. The RTSORAC extensions to the Open OODB architecture are designed within Open OODB's original framework, as shown in Figure 9.

There are two changes to the system's underlying architecture: implementation of extensions using a real-time POSIX operating system [27]; and incorporation of a real-time

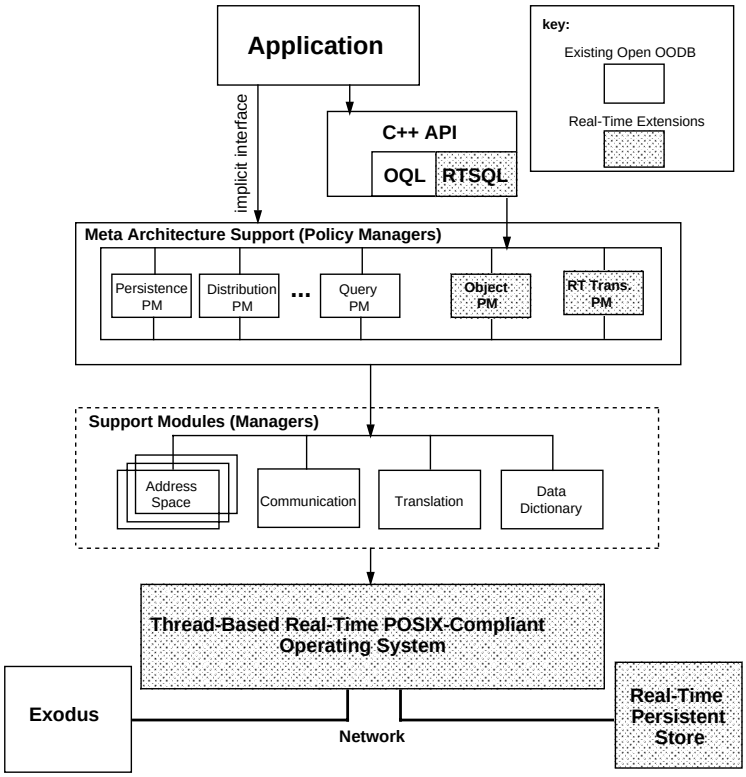

Figure 9: Open OODB Architecture With Real-Time Extensions

persistent storage subsystem, which is the Zip RTDBMS described in Section 5.1. Additionally, two policy managers have been added: one for real-time transaction management, and one for real-time object management.

The interface to the real-time Open OODB prototype is and extension to OQL, based on the real-time SQL language, which is described in Section 5.4. The prototype real-time extensions provide the capability to create arbitrary real-time attribute classes by using the type of the value field as an argument to a C++ template that provides time and imprecision capabilities.

Transactions in the real-time Open OODB are C++ programs that include the schema file of object type declarations. Each transaction is a Real-Time SQL program (see Section 5.4) that is compiled into a POSIX process. Each process maps all database objects, which reside in shared memory into its own address space. The process uses calls to the semantic concurrency control mechanism (Section 6.2.3) to lock objects while using them. These calls are provided by the Open OODB policy manager code. Once an object is locked, the transaction calls the object's methods as if the object were in the transaction's own address space. A transaction process uses calls to the underlying operating system to set its priority and to set alarms for start times and deadlines. The real-time transaction scheduling performed by the Transaction Policy Manager is essentially a mapping of timing constraints expressed in RTSORAC transactions into real-time POSIX priorities for transaction processes. This mapping is designed so that the transaction process priorities realize Earliest-Deadline-First (EDF) scheduling.

Although the Real-time extensions to Open OODB is an academic prototype, it does address many of the requirements of a full real-time database. The use of the RTSORAC model is instrumental in the real-time extensions' support of the requirements of Section 3. The RTSORAC model expresses logical consistency, temporal consistency, and imprecision constraints as well as their trade-offs for both data objects and transactions. It also supports expression of complex data types and associations among data items. The prototype uses main-memory objects with semantic real-time concurrency control to achieve fast access that observes the semantics of the logical, temporal, and imprecision constraints.

5.4 Real-Time SQL

For the past three years organizations including: the University of Rhode Island, the University of Massachusetts - Dartmouth, the U.S Department of Defense's Next Generation Computer Resources (NGCR) Database Interface Standard Working Group (DISWG), and the

American National Standards Institute's (ANSI) Predictable Real-time Information Management Task Group (PRIS-TG), have been working to define real-time database extensions to SQL. These extensions involve timing constraints or data (in SQL data definition), timing constraints on SQL data manipulation, recovery from timing constraint violations, support for predictability, and flexible transaction structure.

Currently, SQL (called SQL2) supports the definition, manipulation, and control of data in a relational database system. However, the current SQL standard (SQL2) [28] has no provisions for real-time database support. The standard does have mechanisms for constraint expression, support for expression of time, and rudimentary transaction structure – all of which provide a basis for developing real-time database extensions.

5.4.1 SQL2 Time Specification.

SQL2 provides sufficient syntax and semantics for specification of timing expressions. There are three *datetime* data types: `DATE`, `TIME`, and `TIMESTAMP`. These data types can be used to express absolute time, such as 9am. There is also an *interval* data type called `INTERVAL`, that can be used to express a period of time, such as 5 minutes. SQL also supports three *datetime* valued functions: `CURRENT_DATE` returns the current date, `CURRENT_TIME` returns the current time, and `CURRENT_TIMESTAMP` returns the current date concatenated with the current time. The arithmetic operators `+`, `-`, `*`, and `/`, and the usual comparison operators (`=`, `<>`, `<`, `<=`, `>`, `>=`) have been defined over *datetime* data types and *interval* data types. Temporal SQL2 [29] has proposed a precise definition for representation of time within the database. This definition includes the concept that time has a discrete representation, and that the smallest unit of time is called a chronon.

5.4.2 Data Temporal Consistency

Constraints in SQL2 are mechanisms for specifying the logical consistency requirements of data. They can be specified on columns, tables, and as stand alone entities (called assertions) within the database. For example, a constraint can be used to specify a range of values for a data item, or to make sure that a foreign key in one table corresponds to a primary key in another.

Specification of Data Temporal Consistency Constraints. In RTSQL, data constraint definitions are extended to allow for specification of temporal consistency requirements of the data. These requirements are usually expressed by indicating the maximum

acceptable age for a data item. Computation of the age of a data item requires that the system record the time that the data value was determined (perhaps it is the time the value was generated by a sensor, or the time that the value was written). Since it would not be necessary to determine the age of every data item in the database, the following RTSQL clause can be specified during definition of a data item to specify that a timestamp will be required:

```
<data timestamp clause> ::=
  [WITH TIMESTAMP <datetime type>]
```

Note that `<datetime type>` is a type provided by SQL2. Access to this timestamp value is through a function on the data item called `TIMESTAMP`. For example, the timestamp value on the data item `temp_reading` would be accessed as `TIMESTAMP(temp_reading)`. Also note that SQL2 provides a function that returns the current time called `CURRENT_TIMESTAMP`. SQL2 provides syntax for constraint specification that can be used to specify temporal consistency requirements when used in conjunction with `CURRENT_TIMESTAMP` and the `TIMESTAMP` function. For example, the following constraint, `temp_reading_avi`, specifies that for the data item `temp_reading` to be absolutely temporally consistent, it must be less than ten seconds old:

```
CONSTRAINT temp_reading_avi
  CHECK (CURRENT_TIMESTAMP - TIMESTAMP(temp_reading)) DAY to SECOND
          < INTERVAL '10' SECOND
```

Here, the SQL2 `CHECK` clause contains a boolean expression which computes the age of `temp_reading` and determines whether it is less than 10 seconds old. SQL2 does not allow constraint specifications to include references to any of the functions that return dates and times (such as `CURRENT_TIMESTAMP`). From the example shown, it is obvious that this restriction must be relaxed in RTSQL.

Relative temporal consistency among data items can be expressed by comparing their timestamps. For example, the following constraint `speed_bearing_rvi` specifies the relative temporal consistency requirements of `speed` and `bearing`:

```
CONSTRAINT speed_bearing_rvi
  CHECK TIMESTAMP(speed) BETWEEN
        TIMESTAMP(bearing) - INTERVAL '2' SECOND
          AND TIMESTAMP(bearing) + INTERVAL '2' SECOND
```

Here, the `speed` timestamp is checked to see if it is within two seconds of the `bearing` timestamp.

Constraints themselves may be valid only for a given period of time. The following RTSQL clauses can be specified as part of a constraint specification to indicate when a constraint is active:

```
<constraint validity interval clause> ::=
   [AFTER <datetime value expression>]
   [BEFORE <datetime value expression>]
```

For example, the previous `speed_bearing_rvi` constraint is specified to be active only after `CONTACT_MADE`, where `CONTACT_MADE` is of a *datetime* data type:

```
CONSTRAINT speed_bearing_rvi
  CHECK TIMESTAMP(speed) BETWEEN
        TIMESTAMP(bearing) - INTERVAL '2' SECOND
          AND TIMESTAMP(bearing) + INTERVAL '2' SECOND
  AFTER CONTACT_MADE
```

Note that the `<constraint validity interval clause>` may be applied to any constraint specification.

Data Temporal Consistency Violations. One of the SQL working groups, SQL2/PSM, is developing support for *condition handling*. Condition handling allows a more active response to the completion of an SQL statement. When a statement is executed, it will either raise an *exception* condition or a *completion* condition. Data temporal consistency constraint violations in RTSQL are detected when a data value is read. This means that the exception will be raised by a statement that may have a corresponding exception handler available. Such a constraint violation could occur if the sensor supplying the data malfunctions, or the transaction responsible for the update misses its deadline. An exception handler could attempt to update the data value or it may simply signal the user that a sensor check should be performed.

5.4.3 Timing Constraints on Execution

SQL2 does not provide any mechanisms for placing timing constraints on statements or transactions. RTSQL specifies time constrained execution by placing timing constraints on individual data manipulation statements or, as appears in SQL2/PSM, a block of statements.

Specification of Execution Timing Constraints. Specification of execution timing constraints uses the following clauses:

```
<timing constraint clause> ::=
  [START BEFORE <datetime value expression>]
  [START AFTER  <datetime value expression>]
  [COMPLETE BEFORE <datetime value expression>]
  [COMPLETE AFTER  <datetime value expression>]
  [PERIOD <interval value expression>
    [START AT <time expression>]
    [UNTIL <boolean expression>]]

<datetime value function> ::=
  CURRENT_DATE | CURRENT_TIME[<time precision>] |
  CURRENT_TIMESTAMP[<timestamp precision>]
```

The `START BEFORE` and `COMPLETE BEFORE` clauses are used to express the latest start time and latest finish time for the execution of the statement. The `START AFTER` and `COMPLETE AFTER` clauses are used to express the earliest start time and earliest finish time for the execution of the statement. The `PERIOD` clause allows for the establishment of a periodic execution of a statement. The `START AT` portion of the `PERIOD` clause establishes the period frame. The `UNTIL` portion of the clause allows for the specification of the conditions that must be met before periodic execution may terminate.

Recall that in SQL2, *datetime* valued expressions have been defined, and can include references to *datetime* value functions. In RTSQL, if such functions are included in an expression, they are all evaluated before the statement begins execution. Further, all of the occurrences of the *datetime* value functions in a statement will appear to have been evaluated at the same instance of time. This holds true for nested statements, where a compound statement may contain statements or other compound statements.

For example, suppose we have the following:

```
X:BEGIN
   SELECT price FROM stocks WHERE name="Acme"
     COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '30' SECOND;
   -- other computations
END X COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '1' MINUTE;
```

The execution timing constraint on the `SELECT` statement specifies that it must complete execution within 30 seconds. The timing constraint on the compound statement specifies that

it must complete execution within 1 minute. Note that the value of CURRENT_TIMESTAMP will be the same for both timing constraints (since they appear in the same compound statement).

Detecting an execution timing constraint violation can be done through the use of timers. When a statement is encountered, the timing constraints are evaluated, and timers are set for the various types of timing constraints.

5.4.4 Predictability Support

In order to support the predictability requirement of real-time databases, RTSQL introduces a concept called a *directive*. Directives provide information to the database system to facilitate maintenance of the constraints and predictability. Directives differ from constraints in that constraints address the logical and temporal consistency requirements for data and operations. The applications that utilize the database system determine these consistency requirements, which in turn are mapped to constraints. Directives provide additional information to the database system to facilitate maintenance of the constraints and predictable access time to data. As such, directives may involve hardware characteristics. Real-time SQL supports the following directives:

- *Data Storage Location* – The RTSQL storage directives are used to allow programmers the ability to specify where and how a data item or table is to be stored. The following clause, part of a table definition, is used to specify storage requirements:

  ```
  <storage clause> ::=
    [STORE IN <storage type> [AT <location>]]
  ```

  where the domains of `<storage type>` and `<location>` are architecture-dependent. For example: `STORE IN main_memory` could be used in a SQL table definition to specify that the table be stored only in main memory. The `AT <location>` clause could be used to store the table at a particular location in main memory. Fixing the location of the data item can facilitate ensuring predictable access time to it.

- *Data Storage Size* – To allow determination of an upper bound on the time it takes to access a table, the following RTSQL directive clause can be specified during the definition of a table:

  ```
  <table size clause> ::=
     [SIZE UPPER LIMIT <integer>]
  ```

indicating that this is the maximum number of data items that can be in this table.

- *Relative Importance Level* – This directive allows for the specification of the relative importance of an action. A scheduling algorithm may use relative importance of tasks as a parameter in determining scheduling priority of the tasks. Not all systems will utilize this directive, and as such, would be free to ignore this directive upon notification. Also, the semantics of the various levels may vary in different systems, hence the portability of this directive is limited. The importance directive clause is as follows:

```
<importance clause> ::=
  IMPORTANCE LEVEL <importance level>
```

- *Asynchronous Execution* – In real-time applications, it may be useful to notify the system that some actions can be done asynchronously. Specification of asynchronous execution of a statement in SQL3 is done using the following clause:

```
[ ASYNC ( <async statement identifier> ) ]
```

  If left unspecified, the default is synchronous. SQL3 also provides a statement for testing the completion of an asynchronous statement:

```
<test completion statement> ::=
    { TEST | WAIT }
    { ALL | ANY | <async statement identifier list> } COMPLETION
```

  The `TEST` alternative is used to check to see if asynchronous statements have completed execution. If they have not, an exception condition is raised. The `WAIT` alternative is used to wait for the asynchronous statements to complete execution. If the asynchronous statements have already completed execution when the `WAIT` statement is executed, an exception condition will be raised.

- Worst Case Execution Time – This directive specifies the worst case execution time (*wcet*) of a statement. This value is made available to the system by the user, who has determined this value through analysis[3].

6 Real-Time Database Research

The vast majority of research in the field of real-time databases has focused on concurrency control and transaction scheduling. Scheduling transactions in a real-time database involves determining which transactions execute when. Similar to tasks in other real-time systems, real-time transactions have priority and must be scheduled accordingly in order to meet specified timing constraints. However, unlike most other real-time processes, real-time transactions access shared data. Therefore, real-time transaction scheduling must take into account the logical consistency of the data and transactions as well as temporal consistency. That is, concurrency control must be considered when scheduling real-time transactions.

According to Abbott and Garcia-Molina [30], a real-time transaction scheduling algorithm has two components: a policy for assigning priorities and a concurrency control mechanism. This section is broken down to describe recent research in real-time database priority assignment policies and in real-time database concurrency control.

6.1 Priority Assignment Policies

There has been much research towards applying real-time scheduling policies to real-time database transaction scheduling. In [30] a performance evaluation of several scheduling policies is presented. The work presented by Huang, et. al. in [31] points out two factors that are used in scheduling real-time transactions: criticality and deadline. They present a scheduling policy in which each transaction is assigned a priority at the time of arrival based on the factor *relative deadline* divided by *criticalness*, where relative deadline is the difference between the transaction deadline and its arrival time. The work presents a performance evaluation comparing the combined scheduling protocol with two others that use only deadline (Earliest Deadline First) or only criticality (Most Critical First) for assigning priority. The results of these tests indicate that transactions that are scheduled using both deadline and criticality miss fewer deadlines and abort fewer transactions than those scheduled using only one of the two factors.

Haritsa, et. al. [32] present two real-time transaction scheduling protocols that are based on the Earliest Deadline First priority assignment policy. The Adaptive Earliest Deadline (AED) protocol is based on the fact that EDF works best when all or most of the transactions can be scheduled. In the AED protocol, transactions are divided into two groups, HIT and $MISS$. The size of the HIT group is determined by a dynamic control variable called $HITcapacity$. When a transaction arrives in the system, it is assigned a random key value.

It is then inserted into a key-ordered list of the transactions currently in the system. If the transaction's position in the list is less than or equal to $HITcapacity$, it is assigned to the HIT group, otherwise it is assigned to the $MISS$ group. Within the HIT group, the priority ordering is by Earliest Deadline. The priority ordering in the $MISS$ group is random.

The goal of the AED algorithm is to collect the largest set of transactions that can be completed before their deadlines in the HIT group. And then the transactions in the HIT group can be optimally scheduled using the Earliest Deadline priority assignment. In order to reach this goal, the $HITcapacity$ control variable must accurately predict the size of the HIT group. The value of $HITcapacity$ is dynamically updated through a feedback process that examines the hit ratio of the transactions in the HIT group versus the hit ratio of all of the transactions.

A second protocol described by Haritsa, et. al. further enhances the AED protocol to allow a transaction's value to be taken into account. The goal is to maximize the sum of the values of those transactions that commit by their deadlines. The new protocol, called Hierarchical Earliest Deadline (HED), groups transactions, based on their values, into a hierarchy of prioritized buckets. It then uses a protocol similar to AED within each bucket to determine the relative priority of the transactions in the bucket. The transactions within a bucket are ordered based on transaction value. As in AED, the priority ordering within the HIT group in each bucket is Earliest Deadline, but unlike AED the priority ordering within the $MISS$ group is Highest Value rather than random.

6.2 Concurrency Control Mechanisms

Most concurrency control mechanisms for real-time databases adapt non-real-time techniques to take time into account. Both pessimistic (lock-based) and optimistic real-time concurrency control techniques have been proposed. In these techniques, serializability is the chosen correctness criterion. Other researchers have recognized that relaxing the serializability requirement can benefit real-time. By using application specific knowledge, semantic concurrency control techniques can provide increased concurrency. Some of the semantic concurrency control techniques described below benefit real-time implicitly with the added concurrency that is available. Others explicitly take temporal consistency into account to further enhance the benefit of using semantics.

6.2.1 Lock-Based Concurrency Control.

Many of the lock-based real-time concurrency control techniques are based on the traditional two-phase locking technique [16]. The techniques, described in [33] and [5], combine two-phase locking with the priority ceiling real-time scheduling algorithm to handle the priority inversion problem. The technique presented in [33] assumes that all transactions are periodic. The scheduling algorithm assigns higher priority to transactions with shorter periods. In this version of the protocol, all locks are exclusive. The *priority ceiling* of a data object is the priority of the highest priority transaction that may lock the object. When a transaction T requests a lock on object O, if the priority of T is not higher than the priority of the data object with the highest priority ceiling of all data objects currently locked by transactions other than T, then the lock is denied. Otherwise the lock is granted. If a transaction blocks a higher priority transaction, it inherits the higher priority while the blocking occurs. When this protocol is combined with a two-phase locking scheme, the resulting concurrency control mechanism bounds blocking to at most one lower priority transaction.

In [5], the protocol of [33] is modified to produce the *Read/Write Priority Ceiling Protocol*. The new protocol allows read/write locking and defines three parameters for each data object in the database. The *write priority ceiling* of a data object is the priority of the highest priority task that may write to the object. The *absolute priority ceiling* of a data object is the priority of the highest priority task that may read or write the data object. The *r/w priority ceiling* of a data object is set dynamically. A task cannot read/write-lock a data object unless its priority is higher than the highest r/w priority ceiling of data objects locked by tasks other than itself. Since the highest r/w priority ceiling of the locked data objects represents the highest priority level at which the currently active transactions can execute, the protocol ensures that a transaction executes at a priority level higher than all preempted transactions.

In [34] Huang, et. al. point out that the priority ceiling protocol [5] requires prior knowledge about the data objects to be accessed by each transaction. They find this requirement too restrictive and present a new protocol that uses priority inheritance combined with priority abort for transaction scheduling. In the new protocol, called Conditional Priority Inheritance, when a priority inversion is detected, if the lower priority transaction is near completion, it inherits the priority of the high priority transaction. This avoids wasting the resources that would result in aborting the nearly complete transaction. If the low priority transaction is not near completion, it is aborted, avoiding the long blocking time for the high priority transaction.

The key to the Conditional Priority Inheritance algorithm is determining when a transaction is nearly complete. A threshold value h is derived so that if the amount of work left to be done by a transaction is less than h, priority inheritance is applied, otherwise priority abort is applied. The sensitivity of h on performance was studied through experiments and the results indicate that with respect to deadline miss ratio, the algorithm performs best for $1 < h < 3$.

Abbott and Garcia-Molina present the *2PL-HP* (two-phase locking with high priority) protocol in [30]. In this protocol, conflicts are resolved by aborting lower priority transactions. If a transaction requesting access to shared data has a higher priority than all other transactions holding locks on the data, the lock holders abort and the requester gets the lock. Otherwise the requester waits for the holder to release the lock.

Another variation of the priority abort idea, called *H2PL* (Hybrid Two-Phase Locking), is presented by Hung and Lam in [35]. In this technique certain conditions, such as transaction workload, are checked to avoid unnecessary aborts. Also, whenever a lower priority transaction that is blocking a higher priority transaction aborts and therefore has to be restarted, its priority is raised to that of the higher transaction to prevent priority inversion (*priority inheritance*). In H2PL, when a transaction requests a lock and a conflict is detected, if the two transactions have the same priority, the requesting transaction is aborted and restarted. In the case where the requesting transaction has higher priority than the transaction holding the conflicting lock, if either the waiting requester or a restart of the holder would miss its deadline, then one of them must be aborted. If the holder is being blocked, it will be aborted, otherwise, the transaction that is further from completion is selected for abortion. Whenever a lower priority transaction is restarted, its priority is raised to that of the higher priority requesting transaction in order to avoid priority inversion. If the requesting transaction has lower priority than the holding transaction, the transaction that is further from completion is aborted and restarted.

The results of performance tests indicate that H2PL performs well for different types of transactions because it is capable of giving preference to higher priority transactions while minimizing the impact on lower priority transactions.

In [36] Lin and Son recognize that the serialization order produced by a concurrency control algorithm should reflect the priority of the transactions. They present a protocol in which the serialization order of active transactions is adjusted dynamically, making it possible for transactions with higher priority to be executed first. Thus higher priority transactions are never blocked by uncommitted lower priority transactions, while lower priority transactions

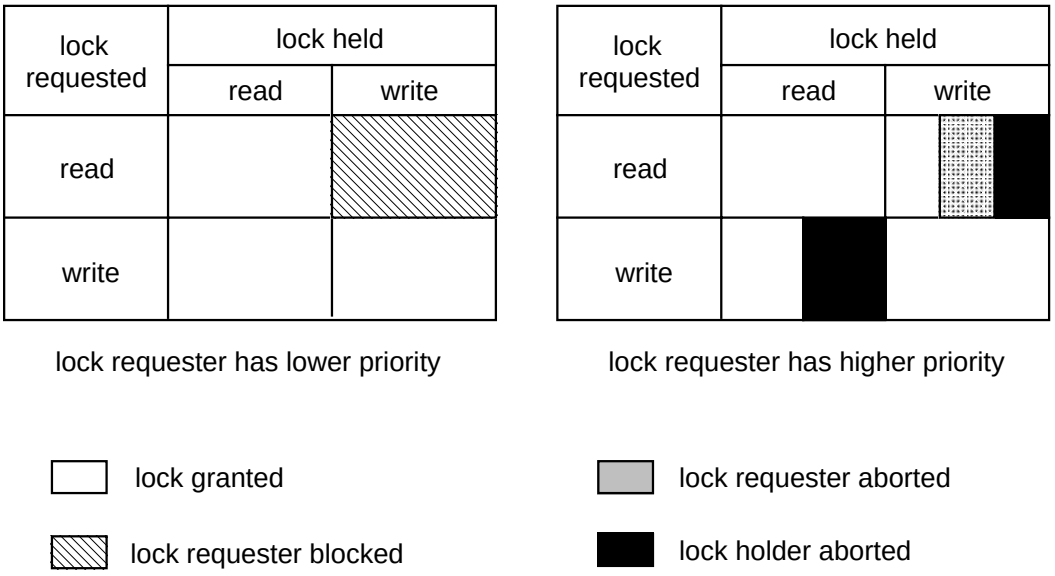

Figure 10: Lin/Son Lock Compatibility Table

may not have to be aborted due to conflicts.

Each transaction is assigned a priority based on its deadline and its start timestamp. The execution of each transaction is divided into three phases: the *read phase*, the *wait phase* and the *write phase*. In the read phase, the transaction acquires read locks on data items and performs *pre-writes* in its own local workspace. The locking protocol is based on the principle that higher priority transactions should complete before lower priority transactions. Figure 10 shows the lock compatibility tables for this protocol. The compatibility depends on the priority of the transactions in question, as well as the types of locks. The wait phase of a transaction allows it to wait until it is allowed to commit. A transaction can commit only if all transactions with higher priority that must precede it in the serialization order are either committed or aborted. A transaction in the wait phase is aborted if it conflicts with a lock request by a higher priority transaction, or if a higher priority transaction that must precede it in the serialization order has already committed. At the end of the wait phase, if the transaction has not aborted, it is assigned a final timestamp and it is committed. In the write phase, a transaction writes all of its changes permanently to the database. The data manager receives write requests for each data object in ascending timestamp order.

6.2.2 Optimistic Concurrency Control

A study of real-time concurrency control techniques in [37] indicates that in systems in which late transactions are discarded, a real-time optimistic concurrency control mechanism outperforms the pessimistic technique of [30]. In [38], a real-time optimistic concurrency protocol called WAIT-50 is presented. In this protocol, a lower priority transaction waits at validation time for any conflicting higher priority transactions to give the higher priority transactions a chance to meet their deadlines first. A wait control mechanism monitors transaction conflict states and dynamically decides when and how long a low priority transaction should wait for its conflicting higher priority transactions.

A real-time optimistic concurrency control technique called OCC-TI [39] uses timestamp intervals to detect conflicts. Every transaction is assigned an initial timestamp interval of $[0, \infty)$. The interval is adjusted to represent serialization order dependencies. A final timestamp is assigned from the interval at the end of the validation phase. The validation of a transaction consists of adjusting timestamp intervals of concurrent transactions and restarting conflicting transactions whose intervals cannot be adjusted. This technique uses the concept of dynamic adjustment of serialization order presented in [36].

6.2.3 Semantic Concurrency Control

A semantic concurrency control mechanism utilizes application specific knowledge to increase concurrency, sometimes defining less restrictive correctness criteria than serializability. In some cases, these correctness criteria are somewhat ad hoc, in that they are based completely on the specific semantics of the application. When serializability is relaxed, imprecision can result in the data and in the transactions. Several correctness criteria have been proposed that formalize how to use application semantics and how to manage the resulting imprecision.

Most work in semantic concurrency control can be divided into two categories: *transaction-based semantic concurrency control* and *object-based semantic concurrency control*. Transaction-based semantic concurrency control capitalizes on the semantics of the known transactions in the system to allow interleavings that might not be allowed in a traditional scheme. Object-based semantic concurrency control manages access to each object in the system based on the semantics of the operations defined on the object. Some of the semantic concurrency control techniques described below benefit real-time databases through the added concurrency they provide. Others take a more active role in real-time by using the temporal requirements of the data and transactions as part of the application semantics.

Correctness Criteria. Agrawal, et. al. [40], generalize transaction based semantic concurrency control with a formal method for determining correct schedules. An *atomic unit* of a transaction T_i relative to another transaction T_j is defined to be a sequence of consecutive operations of T_i such that no operations of T_j are allowed to be executed within this sequence. $Atomicity(T_i, T_j)$ refers to the ordered sequence of atomic units of T_i relative to T_j. A schedule of transactions is a *relatively atomic schedule* if for all transactions T_i and T_j, no operation of T_i is interleaved with an atomic unit of T_j relative to T_i.

The authors of [40] recognize that in general, relative atomicity specifications tend to be conservative because not all potential conflicts occur. They expand the class of relatively atomic schedules to include interleavings of operations that do not have any dependencies between them. An operation o_2 directly depends on an operation o_1 if o_1 precedes o_2 and either both operations are in the same transaction or o_1 conflicts with o_2. A *relatively serial schedule* is defined to be analogous to the notion of serial schedules in the serializability theory. A schedule is relatively serial if for all transactions T_i and T_j, if an operation o of T_i is interleaved with an atomic unit U of T_j relative to T_i, then o does not depend on any operation p in U, and any other operation q in U does not depend on o. A schedule is *relatively serializable* if it is conflict equivalent to some relatively serial schedule. This definition provides a formal correctness criterion for transaction based concurrency control. Further, the authors present a method for determining if a given schedule is relatively serial by testing for acyclicity of a directed graph.

In [41] the use of imprecision in databases and in real-time systems is synthesized and formalized through the concept of *similarity*. The authors define new correctness criteria, less restrictive than serializability, based on the idea that data values that are sufficiently close may be interchanged as input to a transaction without undue adverse effects.

Similarity of a data object is defined by the user based on the semantics of the data. Two views of a transaction are similar if and only if every read event in both views uses similar values with respect to the transaction. Two database states are similar if the corresponding values of every data object in the two states are similar. These definitions are used to extend the traditional correctness criteria, final-state serializability, view serializability and conflict serializability to new criteria based on similarity.

Epsilon serializability (ESR) [14, 42] is a correctness criterion that generalizes serializability by allowing bounded imprecision in transaction processing. ESR assumes that serializable schedules of transactions using precise data always result in precise data in the database and in precise return values from transactions.

A transaction t specifies limits on the amount of imprecision that it can import ($import\_limit_{t,x}$) and export ($export\_limit_{t,x}$) with respect to a particular data item, x. For every data item x in the database, a data ϵ-specification ($data\_\epsilon_x$) expresses a limit on the amount of imprecision that can be written to x [42].

The amount of imprecision imported and exported by a transaction t with respect to data item x, as well as the imprecision written to x, is accumulated during the transaction's execution through $import\_imprecision_{t,x}$, $export\_imprecision_{t,x}$ and $data\_imprecision_x$ respectively.

ESR defines $Safety$ as a set of conditions that specifies boundaries for the amount of imprecision permitted in transactions and data. Safety is divided into two parts: transaction safety and data safety. Safety for transaction t with respect to data item x is defined in [14] as follows:

$$TR\text{-}Safety_{t,x} \equiv \begin{cases} import\_imprecision_{t,x} \leq import\_limit_{t,x} \\ export\_imprecision_{t,x} \leq export\_limit_{t,x} \end{cases}$$

Data safety is described informally in [42]. It can be formalized for data item x as follows:

$$Data\text{-}Safety_x \equiv data\_imprecision_x \leq data\_\epsilon_x$$

Therefore, ESR is guaranteed if and only if $TR\text{-}Safety_{t,x}$ and $Data\text{-}Safety_x$ are invariant for every transaction t and every data item x.

OESR (Object-Oriented Epsilon Serializability) [43] takes the general ESR correctness criterion and specializes it for the RTSORAC real-time object-oriented database model.

Data in the RTSORAC model is represented by objects. Safety for an object o is defined as follows:

$$Object\text{-}Safety_o \equiv \forall_{a \in o_A}(a.ImpAmt \leq data\_\epsilon_a)$$

where o_A is the set of attributes of o. That is, if every attribute in an object meets its specified imprecision constraints, then the object is safe.

Transactions in the RTSORAC model operate on objects through the methods of the object. Data values are obtained through the return arguments of the methods and are passed to the objects through the input arguments of methods. Let t_{MI} be the set of method invocations in a transaction t and let o_M be the set of methods in an object o. The method invocations on o invoked by t are denoted as $t_{MI} \sqcap o_M$. Safety of a transaction (OT) t with respect to an object o is defined as follows:

$$OT\text{-}Safety_{t,o} \equiv \begin{cases} \forall_{m \in (t_{MI} \sqcap o_M)} \forall_{r \in ReturnArgs(m)}(r.ImpAmt \leq import\_limit_r) \\ \forall_{m \in (t_{MI} \sqcap o_M)} \forall_{i \in InputArgs(m)}(i.ImpAmt \leq export\_limit_i) \end{cases}$$

That is, as long as the arguments of the method invocations on object o invoked by OT t are within their imprecision limits, then t is safe with respect to o.

Thus, Object Epsilon Serializability (OESR) is guaranteed if and only if $OT\text{-}Safety_{t,o}$ and $Object\text{-}Safety_o$ are invariant for every object transaction t and every object o.

Transaction-Based Semantic Concurrency Control. In [44], Garcia-Molina defines a *semantically consistent schedule* to be a schedule that transforms the database into a consistent state. Transactions are classified into semantic types based on what they do in the database. For each type, a *compatibility set* is defined to identify which other types are compatible with, i.e., may interleave with, the given type. The user divides a transaction type into *atomic steps* where a step represents some indivisible, real-world action. Any interleaving that is allowed is between these user-defined steps. When a transaction requires access to a data object, it requests a lock. If no other locks are held on the object, the request is granted and the object keeps track of the compatibility set of the type of transaction holding the lock. If another transaction attempts to lock the same object, the transaction processing mechanism checks to see if the type of the requesting transaction is in the compatibility set of the transaction already holding the lock. If so, the lock is granted, if not, the transaction must wait to gain access to the object. In this technique, serializability is replaced as a correctness criterion by *semantic consistency*.

In [45], Lynch presents an approach similar to Garcia-Molina's [44]. Each transaction has a different set of breakpoints with respect to each different transaction type. This approach allows varying levels of concurrency among different types of transactions. Transactions are grouped into *nested classes*. As the classes become more refined, the level of atomicity becomes finer. For each class, breakpoints inserted in a transaction define where other transactions of the same class may interleave. The breakpoints of higher level classes are carried down to the lower level classes. Therefore, for each transaction t, the set of breakpoints where another transaction t' can interrupt is determined by the lowest class containing both t and t'. The levels of atomicity produced by this technique form a hierarchy of allowable interleavings among transactions.

Another transaction-based semantic concurrency control mechanism is described by Farrag and Ozsu in [46]. This work extends the work of Garcia-Molina [44] and Lynch [45] by creating fewer restrictions on allowable interleavings. Nested classes are not used, and therefore the interleavings are not required to be hierarchical as in [45]. Transactions are classified by types and are divided by placing breakpoints between operations where certain

Lock Requested	Lock Held			
	S	E	RS	RE
S	YES	NO	YES	COND
E	NO	NO	COND	COND

Table 4: Lock Compatibility Table

interleavings are allowed. Each breakpoint has associated with it a set, called the *interleaving set*, containing the types of transactions that are permitted to interrupt at that point. Four kinds of locks are used in the concurrency control technique described: *shared, exclusive, relatively shared* and *relatively exclusive*. A shared lock or exclusive lock is granted in the traditional way for read access or write access respectively. Relatively shared and relatively exclusive locks are used to produce non-serializable interleavings. At a breakpoint, the lock can change depending on the actions taken before that point. A shared lock becomes a relatively shared lock at a breakpoint if there is no update before it, otherwise it becomes an exclusive lock. An exclusive lock always becomes a relatively exclusive lock at a breakpoint. A compatibility table, as seen in Table 4, is given for these four locks and while some of the entries are simply YES or NO, others, labeled COND, depend on whether or not the type of the transaction requesting the lock is in the interleaving set of the type of the transaction holding the lock. Locks are released after termination of the transaction.

The *Similarity Stack Protocol* (SSP) described in [47] defines similarity of data based on the time at which the data is written. Two data items are considered to be similar if their timestamps are within a specified bound. Transactions are placed on a scheduling stack according to their priorities. Read/write events of different transactions may swap positions on the stack as long as they are similar.

Several concurrency control techniques have been designed to maintain Epsilon Serializability [14]. Wu et. al. [48] describe several concurrency control techniques in which read-only transaction need not be serializable with other update transactions, but update transactions must be serializable among themselves. The techniques are variations of two-phase locking, timestamp ordering and optimistic concurrency control. The concurrency control protocols presented by Pu et. al. in [49] extend the notion of epsilon serializability to distributed databases. They allow divergence from consistency among database sites as long as their differences remain within specified limits.

Object-Based Semantic Concurrency Control The techniques described in this section take advantage, to varying degrees, of the opportunity for increased concurrency provided by the object-oriented paradigm.

Badrinath and Ramamritham [50] present an object-based semantic concurrency control technique that is used in a system which allows nested data objects, i.e., objects containing other objects. A hierarchical structure, called a *granularity graph*, is used to represent the nesting. The outermost object is represented at the root of the graph and the children of the root represent the objects nested inside. For each operation defined on the object, an *affected set* is computed, containing all nodes in the graph that are affected by the operation. Concurrency is controlled by avoiding conflicts among the operations on the object. A conflict occurs between two operations if they do not *commute*, that is, if the order in which they are performed affects the results returned by the operations or the resulting state of the object. The approach to determining compatibilities between operations is divided into two steps. First, the semantics of the operations are analyzed to determine if they are always compatible, never compatible or conditionally compatible. The second step is performed dynamically when the operations are requested, to determine the value of a conditional compatibility. This value is determined by computing the intersection of the affected sets of the two operations in question. If this intersection is empty, then the operations commute and therefore are compatible.

Weihl [51] describes another object-based mechanism that uses commutativity as the definition of compatibility. Two slightly different versions of commutativity are defined, *forward commutativity* and *backward commutativity*. The difference between these criteria is subtle and the author asserts that they are both necessary because each one is used with different recovery mechanisms. Forward commutativity is designed to work with intentions lists, while backward commutativity works with recovery using undo logs. One of the major results of this work is that concurrency control and recovery are closely linked and must be considered together. When compatibility between operations is in question, commutativity is computed dynamically, as in [50].

Badrinath and Ramamritham [52] present another technique very similar to their earlier work [50]. In the more recent technique, compatibility between operations is based on *recoverability* and not on commutativity. An operation, o_1 is recoverable relative to another operation, o_2, if the outcome of performing o_2 is the same whether or not o_1 executed immediately before o_2. Recoverable operations are allowed to execute concurrently but must commit in the order in which they were invoked.

The three object-based semantic concurrency control techniques described above add concurrency to a database by exploiting the semantics of the object's operations, but each ultimately requires serializability as a correctness criterion. Other researchers have increased concurrency even further by relaxing the serializability constraint. In both [22] and [53] the database designer defines the compatibility between operations on an object. This user-defined compatibility may or may not preserve serializability. Consistency constraints are determined by the designer and implemented through the compatibility relations.

In [22], Wolfe, et. al. present RTC, a language to control real-time concurrency. Objects called *resources* have actions defined on them. The *compatibility relation* C_r is a non-symmetric relation on these action that determines if two actions are compatible. That is, if the actions can be overlapped to result in a consistent state of the resource. The designer of the system must ensure the correctness of the compatibility relation with respect to the semantics of the resource being defined.

In the work of Schwartz and Spector [53] as well, the user is responsible for defining compatibilities, but the authors present some guidelines for doing so. The user defines all possible dependencies among the operations of an object, possibly involving values of parameters. Some of these dependencies are characterized as insignificant in that cycles formed by them do not affect data consistency. Rather than using serializability as the correctness criterion, a schedule is considered correct if it is orderable with respect to a relation formed by combining all of the significant dependencies in the objects involved.

Schwartz and Spector also present the concept of a type-specific locking protocol. The locks that a transaction requests should be held only as long as the semantics of the application suggest. Therefore, each application will use a type-specific locking protocol to determine when locks should be released.

In [54], Wong and Agarwal present another concurrency control protocol that allows bounded inconsistency. The protocol works on an object-based model. In this model, a transaction invokes an operation on an object and the object has a set of possible actions, called the *resolution set*, from which to execute the operation. The state of an object is defined by the sequence of resolutions that have been performed in response to invoked operations. Two resolution sequences are considered equivalent if the resulting object states are the same.

The object designer determines compatibility of object operations based on the notion of *commutativity* with bounded inconsistency. For every resolution sequence $o_p.o_q$ of the operation sequence $p.q$, the designer defines a *forward resolution set dilating function* (f_{pq})

and a *backward resolution set dilating function* (b_{pq}). These functions are defined such that for every state s, if there is a resolution sequence $o_p.o_q$ of the operation sequence $p.q$ with o_p in the resolution set of p ($rs(p)$) and o_q in the resolution set of q ($rs(q)$), then there exists a resolution sequence $o'_q.o'_p$ that is equivalent to $o_p.o_q$ for the operation sequence $q.p$ with o'_p in $f_{pq}(rs(p))$ and o'_q in $b_{pq}(rs(q))$.

The resolution set dilating functions are placed in a compatibility table. When a transaction invokes an operation on an object, the concurrency control mechanism looks in the table and evaluates the resolution set dilating functions to determine if the invoked operation is compatible with all concurrent operations in the object. If the operations are found to be compatible, the resolution sets of the corresponding operations are updated to take into account any inconsistency that may have been allowed by the interleaving of operations. The object designer specifies inconsistency limits for each operation and the protocol ensures that the limits are not violated.

The concurrency control technique presented by DiPippo and Wolfe in [23, 55] is based on the RTSORAC model (described in Section 4.3). It uses *semantic locks* to determine which transactions may invoke methods on an object. The semantic locking mechanism uses a set of preconditions and the object's compatibility function to determine if a requested semantic lock should be granted.

The compatibility function is a run-time function that evaluates a Boolean expression and is defined on every ordered pair of methods of the object. The Boolean expression is used to determine if the pair of methods involved may execute concurrently. It may contain predicates involving characteristics of the object or of the system in general, such as affected sets of methods, temporal consistency and/or imprecision of the data involved, and values of method arguments.

In the semantic locking mechanism, there are two possible outcomes to a semantic lock request: either 1) the semantic lock becomes active and the associated method invocation is executed, or 2) the request is placed on a priority queue to be granted later. Figure 11 illustrates the steps of the semantic locking mechanism.

The first phase of the semantic locking mechanism computes the potential amount of imprecision that the requested method will introduce into the attributes that it writes and into its return arguments. The next phase of the semantic locking mechanism tests preconditions that determine if granting the lock would violate temporal consistency or imprecision constraints. If any precondition fails, then the semantic locking mechanism places the request on the priority queue to be retried when another lock is released. If the preconditions hold,

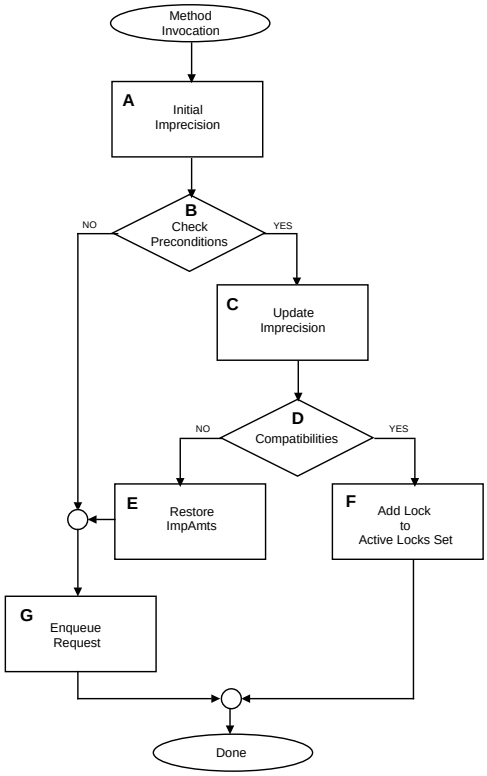

Figure 11: Semantic Locking Mechanism

the semantic locking mechanism updates the imprecision amounts of the data that will be affected by execution of the requested method. Upon successful passing of the preconditions, the semantic locking mechanism checks the compatibility function to make sure that the requested method is compatible with all of the currently active method invocations, as well as all requested method invocations on the queue with higher priority. For each compatibility function test that succeeds, the mechanism accumulates the imprecision that could be introduced by the corresponding interleaving. If all tests succeed, the semantic locking mechanism grants the semantic lock, places it in the active lock set, and makes the requested method ready for execution. If any test fails, the mechanism restores the original values of any changed imprecision amounts and places the request in the priority queue to be retried when another lock is released.

This concurrency control technique allows for the expression of the trade-off between temporal and logical consistency, through the user-defined compatibility function. It has been shown to maintain OESR under certain restrictions, and thus can bound imprecision while making an active effort to meet data and transaction timing constraints.

7 Conclusion

The added dimension of temporal consistency requirements to the requirements of a traditional database complicates the design of real-time databases. Furthermore, the predictability concern of hard real-time applications often requires simplification of database techniques. This paradox has caused real-time database development to lag behind that of non-real-time databases. For this reason the only two commercial real-time databases that we know of, Zip and EagleSpeed, are still far from meeting most real-time database requirements. Recent research in modeling, scheduling, and concurrency control has started to pave the way for better-suited real-time database systems. The potential market for real-time databases in control applications is large. Standardization efforts, like RTSQL, should help further the development efforts.

Despite the recent research and development efforts, there remains a great deal of work to be done to make real-time databases fully meet the requirements outlined in Section 3. Some these questions include: What architecture and operating system support is necessary? How is recovery performed? How can inconsistency be managed and used? How do these requirements impact *active database* design efforts? How does distribution affect real-time requirements? Can such real-time database systems actually be built and used? Can their

interfaces be standardized? The next five years should be important ones in answering some of these questions.

Acknolwedgements. We thank Janet Prichard and Paul Fortier for their efforts in writing the Real-Time SQL subsection, and Roman Ginis for his review of the Zip RTDBM system.

References

[1] J. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *IEEE Computer*, vol. 21, Oct. 1988.

[2] J. Stankovic and K. Ramaritham, "The spring kernel: A new paradigm for real-time operating systems," *ACM Operating Systems Review*, vol. 23, pp. 54–71, July 1989.

[3] W. Pugh and T. M. (Editors), *Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems.* ACM SIGPLAN, 1994.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, 1973.

[5] L. Sha, R. Rajkumar, S. Son, and C. Chang, "A real-time locking protocol," *IEEE Transactions on Computers*, vol. 40, pp. 793–800, July 1991.

[6] R. Rajkumar, *Task Synchronization in Real-Time systems.* PhD thesis, Carnegie Mellon University, 1989.

[7] H. Tokuda and C. Mercer, "ARTS: A distributed real-time kernel," *ACM Operating Systems Review*, vol. 23, pp. 29–53, July 1989.

[8] H. Tokuda, T. Nakajima, and P. Rao, "Real-time mach: Towards a predictable real-time system," in *the USENIX Mach Workshop*, pp. 1–8, 1990.

[9] B. Gallmestier and C. Lanier, "Early experience with POSIX 1003.4 and POSIX 1003.4a," in *IEEE Real-Time Systems Symposium*, Dec. 1991.

[10] J. Senerchia, "A dynamic real-time scheduler for posix 1003.4a compliant operating systems," 1993. Master's Thesis. Dept. of Computer Science, The University of Rhode Island.

[11] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 564–577, May 1987.

[12] W. Zhao, K. Ramamritham, and J. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, vol. C-36, pp. 949–960, August 1987.

[13] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, "Algorithms for scheduling imprecise computation," *IEEE Computer*, vol. 24, May 1991.

[14] K. Ramamritham and C. Pu, "A formal characterization of epsilon serializability,". to appear in *Transactions on Knowledge and Data Engineering*.

[15] K. Ramamritham, "Real-time databases," *International Journal of Distributed and Parallel Databases*, vol. 1, 1993.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. New York: Addison Wesley, 1986.

[17] K. Gordon, *DISWG Database Management Systems Requirements*. Alexandria, Va.: NGCR SPAWAR 331 2B2, 1993.

[18] P. Fortier, J. Prichard, and V. F. Wolfe, "Sql/rt: Real-time database extensions to the sql standard," 1994. To appear in *Standards and Interface Journal*.

[19] S. H. Son, ed., *Advances in Real-Time Systems*, ch. Predictability and Consistency in Real-Time Database Systems, pp. 509–531. Prentice Hall, 1995.

[20] J. Prichard, L. C. DiPippo, J. Peckham, and V. F. Wolfe, "Rtsorac: A real-time object-oriented database model," in *Proceedings of the International Conference on Database and Expert Systems Applications*, Sept 1994.

[21] G. Booch, *Object-Oriented Design*. Redwood City, CA: The Benjamin/Cummings Publishing Company, 1991.

[22] V. Wolfe, S. Davidson, and I. Lee, "*RTC*: Language support for real-time concurrency," *Real-Time Systems*, vol. 5, pp. 63–87, March 1993.

[23] L. B. C. DiPippo and V. F. Wolfe, "Object-based semantic real-time concurrency control," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[24] O. D. Garcia and P. M. D. Gray, *IFIP TC2 Conference on Database Semantics: Object-Oriented Databases*, ch. Semantic-rich user-defined relationship as a main constructor in object-oriented databases, pp. 144–154. North-Holland, 1990.

[25] V. F. Wolfe, L. C. DiPippo, J. P. JJ Prichard, and P. Fortier, "The design of real-time extensions to the open object-oriented database system," in *IEEE Workshop on Object-Oriented Dependable Systems*, Oct. 1994.

[26] D. L. Wells, J. A. Blakely, and C. W. Thompson, "Architechture of an open object-oriented database management system," *IEEE Computer*, vol. 25, pp. 74–82, October 1992.

[27] IEEE, *Portable Operating System Interface (POSIX); Part 1: System API; Ammendment 1: Real-time Extension*. IEEE, 1994.

[28] J. Melton and A. Simon, *Understanding the New SQL: A Complete Guide*. Morgan Kauffman Publishers, 1992.

[29] R. Snodgrass, "TSQL2 language specification," *ACM SIGMOD Record*, vol. 23, pp. 65–86, March 1994.

[30] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," in *14th VLDB Conference*, Aug. 1988.

[31] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental evaluation of real-time transaction processing," in *IEEE Real-Time Systems Symposium*, Dec. 1989.

[32] J. R. Haritsa, M. Livny, and M. J. Carey, "Earliest deadline scheduling for real-time database systems," in *IEEE Real-Time Systems Symposium*, Dec. 1990.

[33] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Concurrency control for distributed real-time databases," *SIGMOD Record*, vol. 17, pp. 82–98, March 1988.

[34] J. Huang and J. Stankovic, "On using priority inheritance in real-time databases," in *IEEE Real-Time Systems Symposium*, Dec. 1991.

[35] S. Hung and K. Lam, "Locking protocols for concurrency control in real-time database systems," *SIGMOD Record*, vol. 21, pp. 22–27, December 1992.

[36] Y. Lin and S. Son, "Concurrency control in real-time databases by dynamic adjustment of serialization order," in *IEEE Real-Time Systems Symposium*, Dec. 1990.

[37] J. R. Haritsa, M. J. Carey, and M. Livny, "On being optimistic about real-time constraints," in *ACM PODS Symposium*, April 1990.

[38] M. J. C. Jayant R. Haritsa and M. Livny, "Dynamic optimistic concurrency control," in *IEEE Real-Time Systems Symposium*, Dec. 1990.

[39] J. Lee and S. H. Son, "Using dynamic adjustment of serialization order for real-time database systems," in *Proceedings of IEEE Real-Time Systems Symposium*, Dec. 1993.

[40] D. Agrawal, J. Bruno, A. E. Abbadi, and V. Krishnaswamy, "Relative serializability: An approach for relaxing the atomicity of transactions," in *Proceedings of the 13th Principles of Database Systems*, pp. 139–149, 1994.

[41] T.-W. Kuo and A. K. Mok, "Application semantics and concurrency control of real-time data-intensive applications," in *Real-Time Systems Symposium*, Dec. 1992.

[42] P. Drew and C. Pu, "Asynchronous consistency restoration under epsilon serializability," Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[43] L. C. DiPippo, *Object-Based Semantic Real-Time Concurrency Control*. PhD thesis, University of Rhode Island, 1995.

[44] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database system," *ACM Transactions on Database Systems*, vol. 8, pp. 186–213, June 1983.

[45] N. A. Lynch, "Multilevel concurrency – a new correctness criterion for database concurrency control," *ACM Transactions on Database Systems*, vol. 8, pp. 484–502, December 1983.

[46] A. A. Farrag and M. T. Ozsu, "Using semantic knowledge of transactions to increase concurrency," *ACM Transactions on Database Systems*, vol. 14, pp. 503–525, December 1989.

[47] T.-W. Kuo and A. K. Mok, "*SSP*: A semantics-based protocol for real-time data access," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[48] K.-L. Wu, P. S. Yu, and C. Pu, "Divergence control for epsilon-serializability," in *Proceedings of International Conference on Data Engineering*, 1992.

[49] C. Pu, W. Hseush, G. E. Kaiser, K.-L. Wu, and P. S. Yu, "Distributed divergence control for epsilon serializability," in *Proceedings of 13th International Distributed Computing Conference*, June 1993.

[50] B. Badrinath and K. Ramamritham, "Synchronizing transactions on objects," *IEEE Transactions on Computers*, vol. 37, pp. 541–547, May 1988.

[51] W. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Transactions on Computers*, vol. 37, pp. 1488–1505, Dec. 1988.

[52] B. Badrinath and K. Ramamritham, "Semantics-based concurrency control: Beyond commutativity," *ACM Transaction on Database Systems*, vol. 17, pp. 163–199, March 1992.

[53] P. M. Schwartz and A. Z. Spector, "Synchronizing shared abstract types," *ACM Transactions on Computer Systems*, vol. 2, pp. 223–250, 1984.

[54] M. Wong and D. Agrawal, "Tolerating bounded inconsistency for increasing concurrency in database systems," in *Proceedings of the 11th Principles of Database Systems*, pp. 236–245, 1992.

[55] L. C. DiPippo and V. F. Wolfe, "Object-based semantic real-time concurrency control with bounded imprecision,". To appear *IEEE Transactions on Knowledge and Data Engineering*.