# A Periodic Object Model for Real-Time Systems[1]

H. Rebecca Callison
Department of Computer Science and Engineering
University of Washington, FR-35
Seattle, WA 98195
callison@cs.washington.edu

Technical Report 93-05-02
May 1993

**Abstract**

We introduce time-sensitive objects (TSO's), a data-oriented model for real-time systems. The TSO model imposes time constraints on object values through validity intervals and object histories. Periodic objects, a class of objects within the TSO model, are described in detail and compared with more traditional periodic processes. We identify advantages of periodic objects including greater scheduling independence, more opportunity for concurrency, and tolerance of timing faults.

[1]This research was supported in part by the Office of Naval Research under grant number N00014-89-J-1040 and by a Graduate Research Fellowship from the Clare Boothe Luce Foundation.

## 1 Introduction

Timing constraints for real-time systems are typically expressed with respect to processing, for example, period, release time, and deadline of a task, and delay [12, 16, 17]. These constraints are often applied in the context of designs based on periodic processing. Working with timing constraints only for *processing*, however, complicates some aspects of real-time system design. Data dependencies between processes with different periods lead to timing interactions that influence the structure of the system and its scheduling flexibility. Low frequency tasks must execute in the intervals when data supplied by high frequency tasks is stable. And high frequency tasks may require fine tuning to make time for the less frequent tasks to execute.

In addition, design based on time-constrained processing provides no *system* framework for recognition of and reaction to timing errors, an important consideration for real-time systems. Timing fault detection is localized to the process in which the error occurs. Timing errors may occur at any point in the execution of a process, and the process structure provides no inherent assistance for regaining the consistent state necessary even for local recovery. Notifying other processes of the timing error takes more time and further complicates the recovery activity.

In this paper we introduce the *time-sensitive object (TSO)* model, a data-oriented alternative to real-time systems design based on time-constrained processing. This model takes advantage of the temporal properties of *data*, including its periodicity, in addition to the timing characteristics of processing. We focus on *periodic objects*, a class of time-sensitive objects, as a basis for comparison with the traditional periodic processes. These objects provide a framework for reducing deadline interactions among processing related by data dependencies, increasing concurrency, and improving tolerance to timing faults.

Following a brief description of the general TSO model, we describe the operational concepts for periodic objects in some detail. In Section 3 we present benefits of designs based on periodic objects through comparison with a periodic process model. Related work is reviewed in Section 4. Section 5 concludes with a discussion of issues and ongoing work.

## 2 Time-Sensitive Objects: An Informal Description

A distinguishing feature of many real-time objects is that whatever value the object has at any point in time, that value will no longer be valid in a few milliseconds or seconds. In short, the value of these objects is *time-sensitive*: as time passes the value of the object is expected or even required to change. Examples of such time-sensitive real-world objects include the positions of aircraft in an airspace and the temperature in a containment vessel during a nuclear or chemical reaction.

Examination of objects in a variety of real-time applications reveals a range of time-sensitivity shown in the taxonomy of Figure 1. At the extremes in the classification are single-interval transient and immutable objects. Single-interval transient objects are created, remain in the system for a short time, and disappear without changing. A report from a sensor such as a radar is typical of this

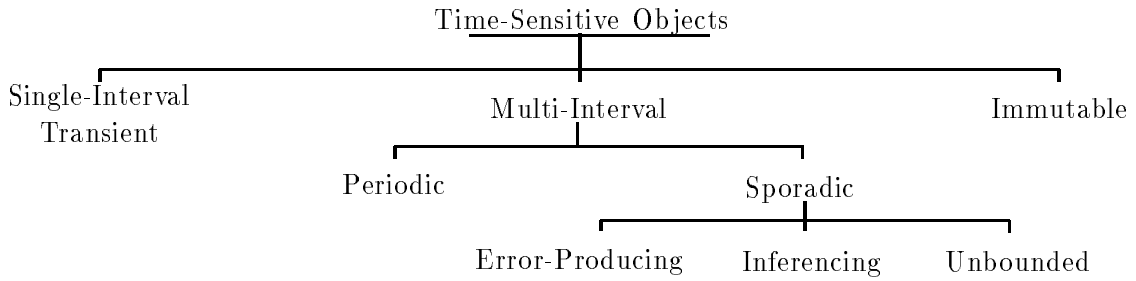



Figure 1: Taxonomy of Time-Sensitivity

class. If it is not recognized and used quickly, its value to the system is lost, often because it has been superceded by a new report. At the other extreme are immutable objects which persist for the life of the system execution without changing.

Most objects in real-time systems fall between these two poles. They persist for some finite length of time and are allowed to change during this lifetime. The expectation of change is captured by associating a validity interval with each successive value assumed by the object, so these objects are called *multi-interval* objects. A validity interval describes the time at which a value became valid and the latest time that it will remain valid, i.e., the time by which the object value is expected to change. Multi-interval objects may be periodic or sporadic. The value of a periodic object is normally reevaluated at fixed temporal intervals with each new value remaining valid for a fixed length period. Sporadic objects are allowed and/or expected to change but not at precisely regular time intervals. For sporadic objects, the absence of change during a validity interval may signal an error, as in a device malfunction. Or it may signal a non-erroneous change of state, as may be inferred, for example, by monitoring the duration of mouse button depressions to distinguish single and double clicks from other actions. A mutable non-real-time object is representable within this classification hierarchy as a sporadic object whose current value may persist for an unbounded time.

A model for the description of systems composed of TSOs is detailed in [4], and a semantics for the interaction of objects within this model is presented. The remainder of this paper is limited to a discussion of periodic objects as a basis for comparison with periodic process oriented designs. For simplicity we use the phrase *periodic object model* to refer to the description and behavior of periodic objects within the broader TSO model.

## 2.1 Periodic Objects: Basic Concepts

A periodic object is a multi-interval object encapsulating past, current, and estimated future object state and exporting operations on these state values. The history of a periodic object comprises a series of values assigned as object state in successive temporal intervals. Each interval consists of (1) an internally consistent state value for the object; (2) the time at which the value became (or is scheduled to become) valid; (3) the time at which the value ceased or will cease to be valid;

and (4) a quality index denoting the "goodness" or precision of the computation that produced the value. Current value is described by an interval which includes the current time as indicated by the system real-time clock. The basic intervals characterizing periodic object value are of equal length and are referred to as *periods*. While new values may be produced for the current period or predicted for future periods, history may not be modified. Time-referenced *read* operation(s) provide access to current, historical, and projected future values. The model for periodic objects also includes three types of operations for modifying object state: *periodic update*, *extrapolate*, and *asynchronous update*.

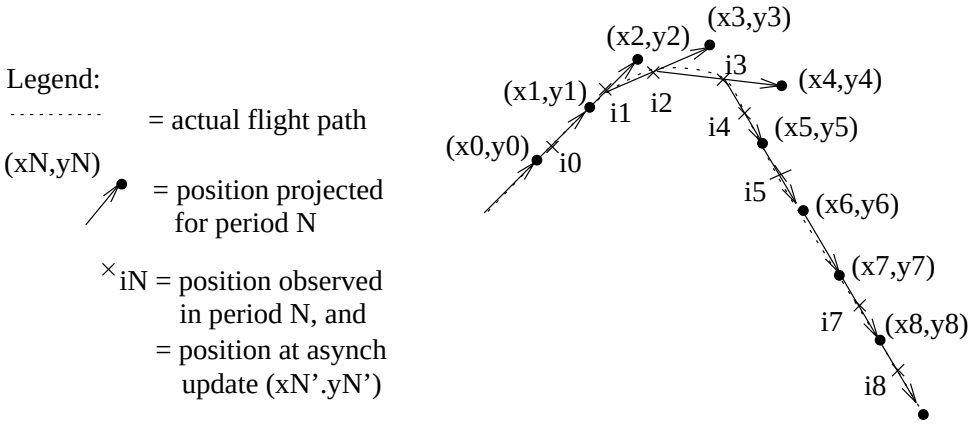
A *periodic update* operation uses new system inputs, evolving system state, and current object value to project the value of the object into the next (upcoming) period. When the current period ends, this projection becomes the new current value. The periodic update is the work horse of the periodic object concept. It is expected to execute in each period to ensure that the time-sensitive value of the object keeps pace with changes in the system environment.

Every periodic object must also possess an *extrapolate* operation. This operation characterizes the expected transformation of the object value over time *in the absence of new inputs*. The extrapolate operation projects the value of the object into a new period based solely on its value in the preceding period(s), i.e., without reference to any system data other than the object's own history. Extrapolation can be thought of as being purely time-triggered: when no new object value has been projected as a new period is entered, the extrapolate operation executes, conceptually instantaneously, to estimate a new current value. In practice, the extrapolate operation could execute (1) in the current period, producing a value that will be overwritten by any normal periodic update completing in the period, (2) on a time-triggered basis as a new period is entered, or (3) asynchronously on demand when an application attempts to access data in a period for which no value has yet been computed.

Sometimes an object value changes unexpectedly in a way not anticipated by the time-sensitive operations of periodic update and extrapolate. To capture this behavior, the periodic object exports a third type of operation for modifying object state, the *asynchronous update*. This operation registers an acceptable but unusual and temporally unpredictable change to the object's value. The new value need not depend in any systematic way on the object history or current value. The value assigned through asynchronous update becomes effective immediately. Its validity interval ends at the end of the current period. Periodic updates and/or extrapolate then use this new value to project time-sensitive components into the future.

An implied contract exists between the periodic object and its clients. Each periodic object is conceptually active, possessing one or more threads of control which are scheduled periodically to accomplish the desired periodic updates. The periodic object makes a best effort attempt to provide new versions of object value on a specific periodic schedule. Further, the object agrees to notify clients of unexpected (asynchronous) updates should they occur. Normally, this notification will take the form of a new version of object state appended to the object history and valid until the end of the current period.[2] The client agrees to use a supplied value only within the constraints

[2]The TSO model also allows a client object to declare a strong dependency on the object, registering it for imme-




(a) Aircraft Flight Path

| Input: | | i1 | | i2 | | i3 | | i4 | | i5 | | i7 | i8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time → | | ↓ | | ↓ | | ↓ | | ↓ | | ↓ | | ↓ | ↓ |
| Period | 1 | 2 | | 3 | | 4 | | 5 | 6 | 7 | 8 | | |
| *x-coord* | x1 | x2 | x2′ | x3 | x3′ | x4 | x4′ | x5 | x6 | x7 | x8 | | |
| *y-coord* | y1 | y2 | y2′ | y3 | y3′ | y4 | y4′ | y5 | y6 | y7 | y8 | | |
| *quality* | PU | PU | AS | XA | AS | XA | AS | PU | PU | XP | PU | | |

Quality Legend: (XP < PU < XA < AS)

XP: Extrapolation from periodic update — PU: Periodic update

XA: Extrapolation from asynchronous update — AS: Asynchronous Update

(b) Evolving Track Object

Figure 2: Periodic Track Object Based

implied by its validity interval. Each client is empowered, however, to continue execution in the absence of an expected value, if desired, through authorized use of the extrapolate operation to estimate a new object value.

The use of a periodic object to track the flight of an aircraft is illustrated in Figure 2. In this highly simplified tracking example, the aircraft broadcasts its position at roughly equal intervals. The broadcast is monitored by the system and represented as a periodic object called a Track, here composed of x-coordinate, y-coordinate, and quality. Since air traffic control is interested in where the aircraft is or is about to be, rather than where it has been, the Track has a periodic update operation that uses a linear function of the aircraft's position in the two most recent periods to project the position ahead one period.[3] When incoming reports agree with the projection (periods

diate notification of asynchronous changes to the influencing object. The semantics and timing of the propagation of asynchronous updates to such strongly dependent objects are beyond the scope of this discussion.

[3]The author does not imply that real world tracking algorithms conform in any meaningful way to this example.

1, 5, 7, and 8), the projected value persists as current value. When the vehicle initiates a turn, as shown in periods 2, 3, and 4, the incoming data does not reinforce the estimated position. An asynchronous update to record the correct reported position is initiated immediately in mid-period. This new value becomes the basis for projection of vehicle position into the next period. When no intelligible position report is received in period 6, the omission is masked by extrapolating unreinforced position data. The quality of the new value informs clients of the reporting failure and subsequent invocation of extrapolation.

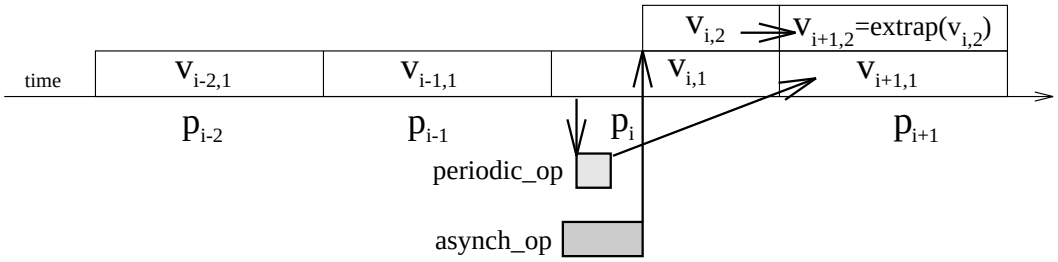
## 2.2 Concurrency Issues

Periodic objects may exist in systems that employ internal concurrency to accommodate concurrency and unpredictable behavior in the system environment. To achieve predictable execution times, reads to current and historical data should never be delayed, implying that the use of locking protocols and/or critical sections is inappropriate for concurrency control.[4] Periodic objects are assumed to be non-blocking and linearizable [10], guaranteeing that (1) some process makes progress in a finite number of steps and (2) that the result of each operation becomes visible atomically at some instant in time between its initiation and completion. Linearizability preserves the order of temporally disjoint operations, but the effective order of operations executed concurrently is nondeterministic. The requirements to maintain access to historical data and to accomplish updates atomically outside of critical sections precludes in-place updating of object value for two reasons: (1) the old value must be maintained and (2) no partial updates may be visible. These characteristics favor an optimistic approach (copy, modify, validate, atomically install new version on success) that leads naturally to a queue structure for retention of history.

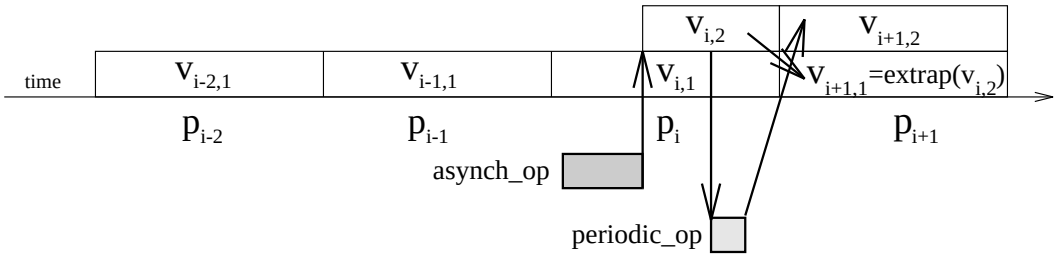
Multiple periodic update operations may be associated with a single object. Any or all of these periodic updates may execute in period i to project the value of the object into period $i + 1$. Each distinct periodic update operation of an object must produce a value of distinct quality ranking. The highest quality periodic update completed in period i must persist into period $i + 1$ to become the new current value. Including quality comparison in the validation step of a projection ensures this characteristic. The results of a lower quality computation are discarded and the computation abandoned rather than overwrite a higher quality result. Allowing multiple periodic updates provides a structure for implementing concepts like imprecise computation [18, 20] and performance polymorphism [13]. Because quality information is available to the object's clients, the application can decide if the value read is sufficiently accurate to be of use.

The concurrency available from non-blocking objects can be further exploited by allowing asynchronous updates to execute concurrently with periodic updates. Potential outcomes arising from executing both a periodic and an asynchronous update on a single object in the same period are illustrated in Figure 3. An asynchronous update conflicts with a concurrent periodic update if it changes the value in the current period while the periodic update is using the newly obsolete value

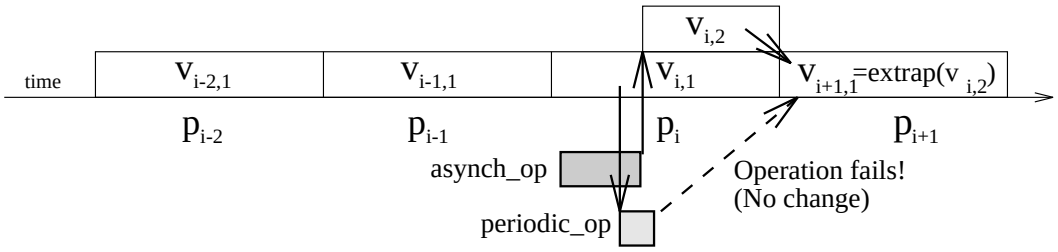
[4] Locking protocols and critical sections allow unpredictable delays. These delays may be unbounded in the presence of deadlock or a failure of a process while holding the lock or inside a critical section.




(a) Periodic update completes before asynch update. Extrapolation of asynchronous update becomes current value in period i+1.




b) Asynch update completes before periodic update begins. Periodic update based on asynchronous change becomes current value in period i+1.




(c) Periodic update begins before asynchronous update completes, but completes after asynchronous update. Periodic update, based on outdated value, never becomes valid.

Figure 3: Resolution of Potential Asynchronous and Periodic Update Conflicts

to project a new value for the next period. If the asynchronous update takes effect after the periodic update, the asynchronous update will be lost when the next interval becomes current. Coupling each asynchronous update atomically with an extrapolate operation based on the new value resolves this conflict. The quality index is enhanced to reflect the priority of both the operation producing the results and the data inputs to the operation. A lower quality operation on "better" data, e.g., extrapolation of an asynchronous, inherently high priority update, will supercede results of a higher quality operation based on "worse" inputs.

A claim for predictable execution of modifications under this scheme is based on a stochastic argument: The type and schedule of operations to be performed routinely are fairly well understood a priori, and the probability for conflict between two modifying operations is very low by design. On the rare occasion when conflict occurs, the application has the options of retrying or abandoning the unsuccessful operation. On average the execution time will be predictable. And the periodic object structure is specifically designed to tolerate the rare timing failure which may result from unexpected conflict. (See further discussion of timing fault tolerance below.)

To limit the amount of process overhead, periodic objects may be aggregated or composed into higher level objects. These higher level objects own the threads of control that execute the operations of the simpler composing objects. This capability decouples the granularity of control (processes) and data (objects) in the system design.

## 3 Comparing Periodic Objects with Periodic Processes

In Section 1 we postulate that real-time systems based on periodic objects will exhibit more robust behavior than systems based on periodic processes by reducing deadline interactions, increasing parallelism, and providing a framework for timing fault tolerance. In order to consider these claims we need first need to introduce simple scheduling models for both periodic processes and periodic objects.

**Periodic Process Model.** Each process is characterized by the tuple $P_i = (s_i, p_i, r_i, c_i, d_i)$ where

$s_i$ = time of beginning of first period of execution of $P_i$
$p_i$ = period length
$r_i$ = offset from beginning of period for release of $P_i$
$c_i$ = maximum units of computation consumed by $P_i$ in each period
$d_i$ = offset from beginning of period of deadline of $P_i$

Assume $c_i, d_i \leq p_i$; and $c_i \leq d_i - r_i$.

**Periodic Object Model** Each periodic object is characterized as $Obj_i = (s_i, p_i, o_i)$ where

$s_i$ = start time of initial validity interval of $Obj_i$,
$p_i$ = length of $Obj_i$ period,
$o_i$ = non-empty set of periodic operations of $Obj_i$; each $op_{i,j} \in o_i$ has characteristics:
$q_{i,j}$ = operation quality,
$r_{i,j}$ = release time for operation, } offset from object period
$d_{i,j}$ = deadline for operation, }
$c_{i,j}$ = maximum computational units for operation execution.
Assume $c_{i,j} \leq d_{i,j} - r_{i,j}$.

The periodic process model describes the frequency, scheduling window, and computational demands of each periodic task. The periodic object model first describes the periodicity of the object value. For generality the model allows for separate description of the timing constraints of each periodic update operation associated with the object.[5]

### 3.1 Minimizing Timing Constraint Interaction

In the process model, data dependencies between processes with different periods introduce explicit dependencies in process scheduling constraints. Sometimes these constraints are artificial: they derive solely from the stability intervals of the data rather than from the functional and timing requirements of the system.

Consider the difficulties encountered in scheduling interactions between two periodic processes $P_0$ and $P_1$. $P_1$ depends on data supplied by $P_0$, and $p_0 \ll p_1$. For simplicity in this example, it is assumed that $s_0 = s_1$ and $p_1 = \mathrm{n}p_0$ for integer n. Because data consistency is not guaranteed during $P_0$ execution, the lower frequency process $P_1$ must run in the "windows of consistency" in the schedule of the higher frequency process $P_0$. A number of scheduling and/or programming modifications may be necessary to fit the two processes into the system schedule as illustrated in Figure 4. If $P_1$ can run to completion between any two executions of $P_0$, the constraints on $P_1$ may be modified as described by $P_1'$, where the release time and deadline are adjusted to select one of the available slots in the $P_0$ schedule:

Case 1. $\mathrm{P}_1' = (\mathrm{s}_1, \mathrm{p}_1, \underline{\mathrm{r}_1' = \mathrm{j} * \mathrm{p}_0}, \underline{\mathrm{d}_1' = (\mathrm{j}+1) * \mathrm{p}_0}, \mathrm{c}_1)$ for some j, $0 \leq \mathrm{j} \leq (\mathrm{n}-1)$

(Underscoring is used throughout this discussion to highlight interprocess and interobject scheduling constraints.) The constant j here selects one of every n $P_0$ periods for execution of $P_1$. If this window is too short for complete execution of $P_1$, the specification constraints on $P_0$'s execution may be tightened to make room for execution of $P_1$ as described by $P_0''$:

Case 2. $\mathrm{P}_0'' = (\mathrm{s}_0, \mathrm{p}_0, \mathrm{r}_0, \mathrm{d}_0'', \mathrm{c}_0)$ where $\underline{d_0'' \leq p_0 - c_1}$, $\mathrm{d}_0'' < \mathrm{d}_0$
$\mathrm{P}_1'' = (\mathrm{s}_1, \mathrm{p}_1, \underline{\mathrm{r}_1'' = \mathrm{j} * \mathrm{p}_0}, \underline{\mathrm{d}_1'' = (\mathrm{j}+1) * \mathrm{p}_0}, \mathrm{c}_1)$ for some j, $0 \leq \mathrm{j} \leq (\mathrm{n}-1)$

[5] Early experience with the model indicates that the number of periodic operations per object will usually be small (two or fewer) and that multiple operations of a single object may have common scheduling constraints.

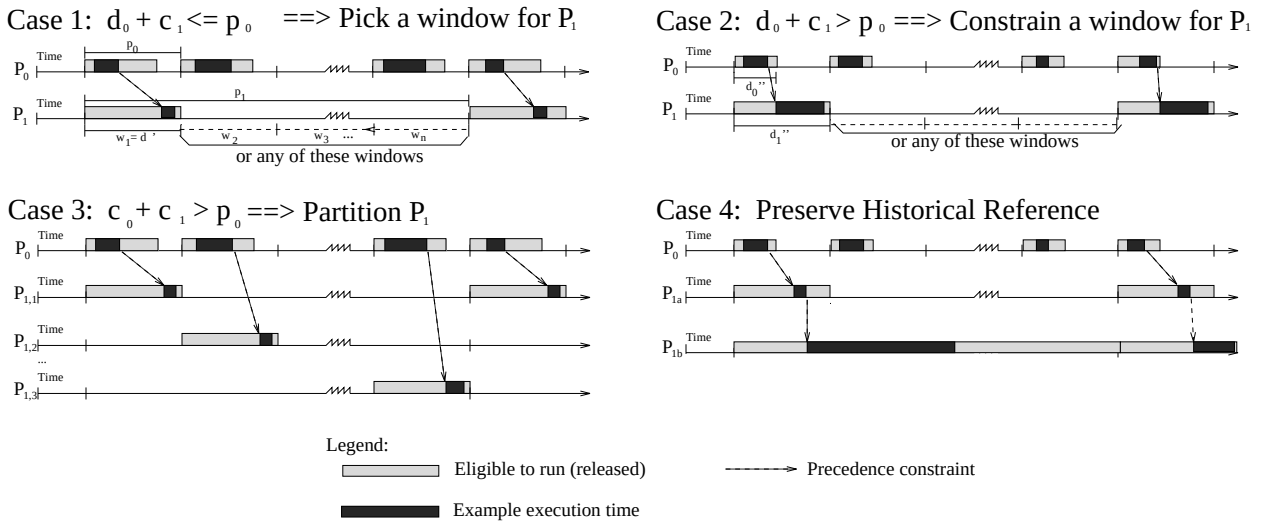



Figure 4: Scheduling Constraints for Data-Dependent Process Pairs

Where no simple constraint on $P_0$ yields a feasible solution, $P_1$ may be partitioned into a series of subprocesses $\{P_{1,0}; P_{1,1}; \ldots; P_{1,n-1}\}$, each of which can run to completion in one of the $P_0$ scheduling windows.[6] Now:

$$\text{Case 3.} \quad P_0''' = (s_0, p_0, r_0, d_0''', c_0) \text{ where } \underline{d_0''' \leq p_0 - \max_j(c_{1,j})}$$
$$P_{1,j} = (s_1, p_1, \underline{r_{1,j} = j * p_0}, \underline{d_{1,j} = (j+1) * p_0}, c_{1,j}), \forall j,\ 0 \leq j \leq (n-1)$$

Alternatively, if $P_1$ depends for success on historically consistent rather than most current $P_0$ data, $P_1$ can be partitioned into a data capture step, $P_{1a}$, and a processing step $P_{1b}$. $P_{1a}$ must run between activations of $P_0$, but $P_{1b}$ is now free of derived constraints except for the precedence constraint imposed with respect to $P_{1a}$ (Case 4 of Figure 4).

This intertwining of scheduling constraints, particularly the interaction between computation time of one process and deadlines of another, has a negative effect on the software design. The partitioning of processes to fit available timing windows is likely to diminish the clarity of the program, complicating development and future maintenance. Schedule constraints become sensitive to timing changes in obscurely related processes. The timing dependencies contribute to tightly coupled system designs which typically require careful tuning throughout the software life cycle.

In the periodic object model, operations synchronize on the availability of desired versions of object value. Again, consider the simple case of a two object dependency in which $Obj_1$ depends on $Obj_0$. Both objects are periodic, each possesses a single periodic operation, and $p_0 \ll p_1$. Synchronization occurs when the desired version of $Obj_0$ state becomes available without concern for other activity

[6] If such partitioning is not possible, a hardware solution may be sought. Moving to a higher performance processor decreases computation time required while holding all other timing characteristics unchanged.

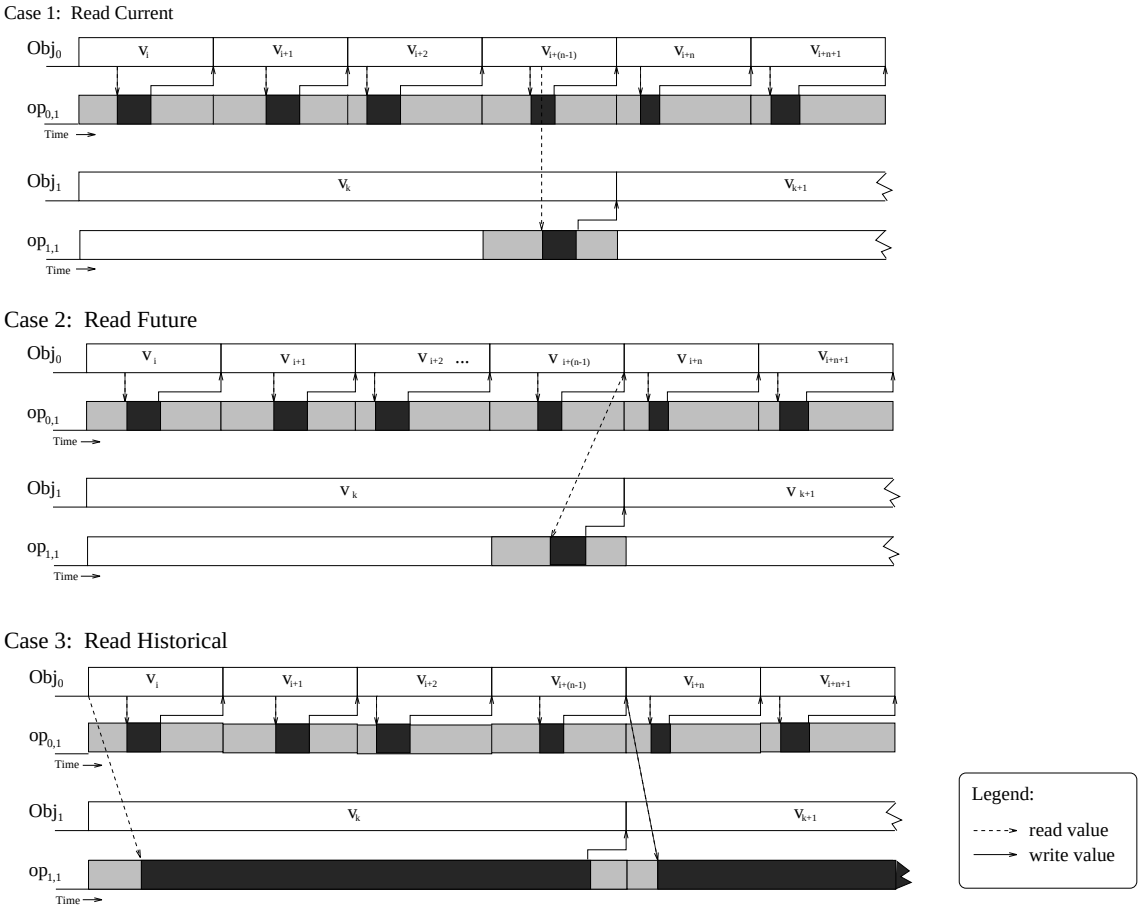



Figure 5: Scheduling Constraints for Dependent Objects

that may be undertaken by the thread (process) executing this operation. Because the threads invoking the two periodic operations may execute concurrently, the deadline of the first need not guarantee execution time for the second. Where correctness of the periodic operation on $Obj_1$ requires use of $Obj_0$ current value, $Obj_1$ operations are explicitly constrained by the *periodicity* of $Obj_0$ data, but not by other execution parameters of $Obj_0$'s operations:

$$\text{per\_op}'_{1,1} = (q_{1,1}, \underline{r'_{1,1} = p_1 - p_0}, d'_{1,1} = p_1, c_{1,1}) \text{ for some j, } 0 \leq j \leq (n-1).$$

A variety of temporal semantics for this interaction are depicted in Figure 5. The application may choose the currently valid value (Case 1) or synchronize on the most timely result by waiting for $Obj_0$'s projection into the upcoming period (Case 2). In Case 3 of Figure 5, where $Obj_1$ depends on $Obj_0$ historical data, there is no interaction of scheduling constraints. The data capture step inserted to avoid scheduling dependencies in Case 4 of the periodic process model is performed automatically through system retention of historical versions.

In contrast with the periodic process model, the periodic object model does not use scheduling constraints to guarantee that operations on dependent objects will complete on time. In Case 2 of Figure 5, where $Obj_1$ waits for data being produced in the current period by $Obj_0$, failure of the periodic update of $Obj_0$ to complete on time means that the $Obj_1$ operation will also fail. Techniques for achieving tolerance to such errors are discussed in Section 3.3 below.

### 3.2 Enabling Concurrency

Periodic objects have two advantages over periodic processes in allowing concurrency: (1) Because context switches have high overhead, the periodic object model allows fine-grained objects to be aggregated into higher level objects. Using this mechanism, the multiple threads of control implicitly associated with the conceptually active fine-grained objects may be aggregated into coarser control entities (processes). Where data dependencies between large-grained periodic processes require precedence relationships to guarantee data consistency in the process model, large-grained threads in systems built from periodic objects can achieve fine-grained synchronization at the object level.[7] Resolution of conflict at the object level, rather than the process level, creates greater opportunity for concurrency. (2) Retention of object histories directly enables concurrent execution of operations that are producing new object values with operations that use only historical data.

In a single processor environment, the primary benefit of enabling concurrency may be that programs are simpler to write and understand, because they are free from the subtle dependencies of process-precedence-based consistency. In a multiprocessor system, the benefit should include speed up from parallel execution of decoupled operations.

### 3.3 Improving Tolerance to Timing Errors

Fault tolerance has four components: (1) fault detection, (2) damage assessment (fault containment), (3) recovery, and (4) repair and return to service. Timing fault detection has been an active research issue in real-time systems. Mechanisms for the expression and monitoring of constraints on program execution time have been widely investigated, e.g., in [5, 12, 16, 17, 21, 23]. These mechanisms in isolation provide only for timing error detection, and this detection is localized to the erroneous process where a timing exception is typically raised. No inherent assistance is provided for damage assessment, recovery, or return to service. Periodic objects extend the capability for tolerance to timing faults:

**Fault detection.** Non-faulty system behavior is characterized by the expected succession of periodic object values. Finding a value missing in the current period implies that an error has occurred in the production of an expected value. The detection mechanism provided by periodic objects is

[7] Fine-grained synchronization can certainly be achieved through ad hoc application of a variety of techniques within the periodic process model. The benefit of the periodic object model is that this advantage is available systematically, rather than on an ad hoc basis.

global. The absence of data can be detected by any client of the object as soon as the validity interval of the value last recorded expires. There is no delay introduced by the necessity for interprocess communication to notify users of the error. Nor is it necessary for each user to set a watchdog timer in order to infer the absence of the data.

**Damage assessment (fault containment).** Periodic objects contribute in two ways to containment of timing faults. (1) The linearizable characteristic of these non-blocking objects prevents visibility of partial updates. The visible state of the object is always internally consistent, regardless of the point at which a timing exception is recognized. (2) Validity intervals coupled with quality information allow object users to infer the status of object operations. Each user can condition its own processing, including the distribution of its results, in response to degradation in the quality of the data it reads. So the structure enables fault containment at the application level.

**Recovery.** Again, the periodic object mechanisms for direct recovery are two-fold: (1) The extrapolate operation permits gaps in periodic data to be bridged. (2) The periodic update characteristic assures that transient errors will be corrected within a tolerable time, i.e., in the next interval. A succession of sub-par values indicates a more extensive problem, but at least the periodic object structure provides enough information for all users to be aware of such problems.

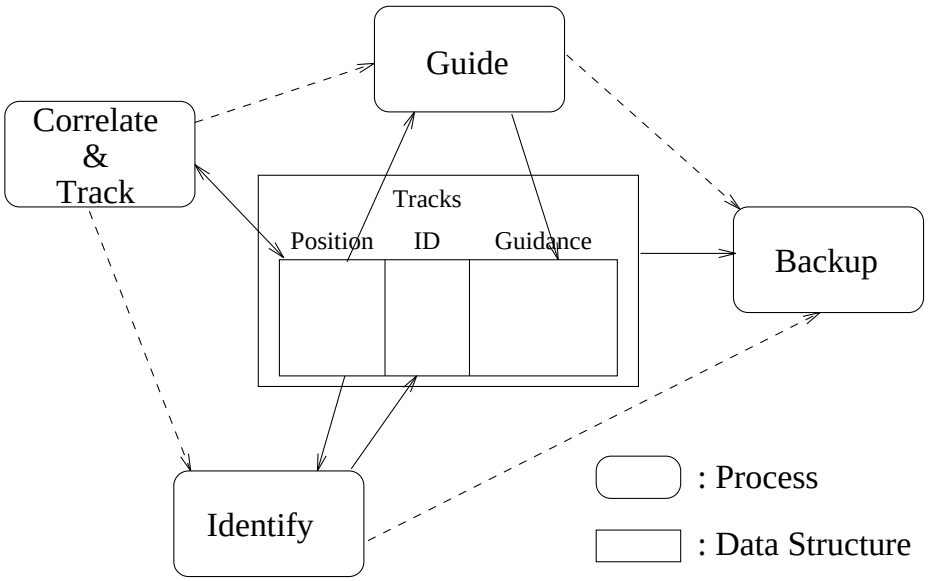
The structure also contributes to successful recovery from more catastrophic failures. Recovery from loss of a processing node or function is possible through extrapolation from historical versions stored on durable media.

**Repair and return to service.** The recovery characteristics described above constitute limited self-repair. Anticipated extensions to the periodic object model will address replicating objects in a distributed environment. This extension may include some system capability to recover lost replicas. It is anticipated, however, that most repair activity will be managed by the application rather than being integral to the periodic object model.

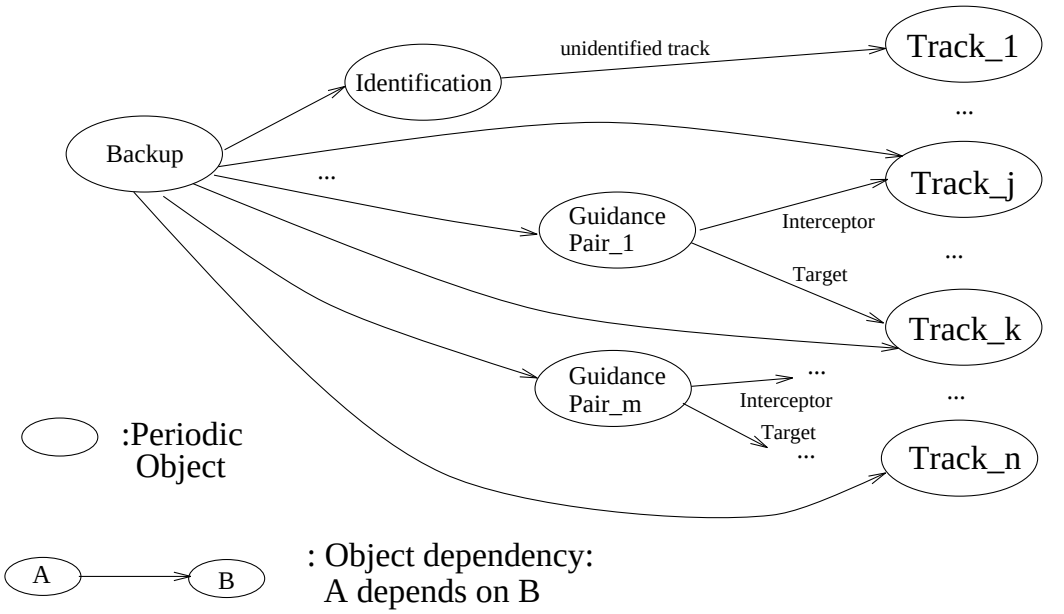
### 3.4 An Air Defense Example

In this section we use an air defense system as a concrete example of some of the problems and resolutions identified earlier. An air defense system is similar to an air traffic control system: The position of aircraft within an airspace are monitored by tracking radar returns from the aircraft. Each aircraft entering the airspace must be identified. When unknown (unidentifiable) or hostile intruders are detected in the airspace, the system provides guidance information (correct heading, speed, etc.) to direct the pilots of interceptor aircraft to a rendezvous with the intruders (targets). Key requirements are presented in Table 1.

Figure 6 presents two alternative designs for the Air Defense system. Figure 6a depicts a traditional, process-based design. In this design all processes operate on a monolithic database of track information. A Correlate&Track process runs to completion in each basic cycle. This process associates incoming radar reports with tracks and calculates new locations for all aircraft. When Correlate&Track completes, Guide and Identify processes compute updated results based on the




(a) Process-based design




(b) Periodic object-based design

Figure 6: Alternative Air Defense System Designs

Table 1: Air Defense System Requirements

| Functional Requirement | Timing Constraint | |
|---|---|---|
| | Type | Seconds |
| **Correlate:** Match incoming radar returns with aircraft tracks | Period | n |
| **Track:** Update location, bearing, speed information for each aircraft | Period | n |
| **Guide:** Calculate heading, speed, direction required for interceptor to reach target | Period | 4n |
| **Identify:** Establish identity of aircraft entering airspace | Sporadic Deadline | 20n |
| **Backup:** Save consistent track database on stable store to support recovery from processor/system failure | Period | 4n |

new (current) track data. The inherently sporadic Identify activity is converted to a periodic process running at the same frequency as Correlate&Track to make sure that it does not miss any critical track position updates. The Backup process must run in isolation to copy a consistent database to stable storage for possible use in fault recovery.

A design based on periodic objects appears as Figure 6b. The basic object of this design is the periodic Track. Each Track instance encapsulates identification and position information for a single aircraft. It has a periodic update operation to project track position based on correlated radar returns. A GuidancePair describes the connection between an interceptor and its target and maintains time-sensitive guidance and interceptor mission data. The GuidancePair depends on the interceptor and target Tracks for current position data. Each Identification instance depends on a single as yet unidentified track object. The timing constraints of the Identification object are not dependent on the period of the track object,[8] because all historical track position data required by Identification is retained in the track history. The Backup object periodically updates stable storage with a consistent view of Track, GuidancePair, Identification, and other system objects based on recent history.

Table 2 contrasts the scheduling constraints imposed by applying these two approaches to Air Defense system design. Precedence relationships dictate scheduling dependencies among all four types of processes in the design of Figure 6a. Guide processing for the whole system is too time-consuming to run in a single Track scheduling window. It has been partitioned into four processes for load balancing across tracking periods. The Correlate&Track deadline is constrained by the need to open a window for execution of Guide, Identify, and Backup. Guide and Identify have no interprocess data dependencies and can run concurrently, but each must allow for the scheduling of Backup.

[8]Identification is specified as a time-sensitive sporadic object, rather than as a periodic object. It is created when a need for track identification is recognized and persists until identification is complete or untilthe deadline for identification is reached.

| Periodic Process Constraints | | | |
|---|---|---|---|
| Process | p | r | d |
| Correlate&Track | $p_{C\&T}$ | 0 | $d_{C\&T} \leq p_{C\&T} - (c_{Id} + \max(\max_i(c_{G_i}), \min_j(c_{G_i}) + c_B))$ |
| Guide$_i$: $\{G_i \mid 0 \leq i \leq 3\}$ | $p_G = 4p_{C\&T}$ | $ip_{C\&T}$ | $i \leq 0 \leq 2$: $d_{G_i} \leq (i+1) * p_{C\&T}$<br>$i = 3$: $d_{G_i} \leq (i+1) * p_{C\&T} - c_B$ |
| Identify | $p_{C\&T}$ | 0 | $d_{Id} \leq p_{C\&T} - c_B$ |
| Backup | $p_B = 4p_{C\&T}$ | $p_B - p_{C\&T}$ | $p_B$ |
| Periodic Object Operation Constraints | | | |
| Object.per_op | p | r | d |
| Track | $p_{C\&T}$ | 0 | $p_{C\&T}$ |
| GuidancePair | $p_G$ | $p_G - p_{C\&T}$ | $p_G$ |
| Identification | – | 0 | $d_{Id}$ |
| Backup | $p_B = 2p_{C\&T}$ | 0 | $2p_{C\&T}$ |

Table 2: Comparison of Scheduling Constraints

In contrast, in the periodic object design the availability of periodic object histories relieves restrictive scheduling constraints on long latency operations of Backup and Identification. These operations may execute concurrently with Track and GuidancePair operations which require current data. Scheduling interaction between Track and GuidancePair objects is simplified, because guidance and track update operations may execute concurrently, synchronizing implicitly when each GuidancePair finds the Track updates it needs to compute a new value.

## 4 Related Work

Time-sensitive objects are related to work in three principal areas:

**Models of time-constrained data.** Data models which include limited validity intervals for data values have been proposed for at least two research systems: MARS [14] and CHAOS [2]. The MARS model requires that all data values be included in messages which have fixed validity times. For fault tolerance, data values which must persist for multiple intervals are broadcast as a new message in each validity interval. Fault tolerance is achieved by requiring that every component of system state be broadcast at some maximum period. Rather than allowing the user to decide if data quality is sufficient for use, the MARS operating system maintains a list of currently active (valid) messages which any client may read. The implication is that a new value always overwrites the old value. Determination of adequate quality is made by the operating system rather than the user of the data. Archival history exists conceptually in MARS but mechanisms for access to archival data by real-time applications have not been elaborated.

CHAOS researchers [7] describe the need for multiple, time-constrained value assignments to ob-

jects, but no mechanisms are published for maintaining or selecting among these versions.

**Real-time databases.** Periodic objects constitute a multi-version, timestamp-based database. The research that resulted in the periodic object concept is motivated by problems for which database concepts provide traditional answers: what techniques can guarantee the recoverability of consistent data even in the presence of system failures? Our earlier examination of the relationship between transaction processing, real-time databases, and time-sensitive objects in [3] cites the following issues with respect to use of transaction processing in real-time: (1) The temporal unpredictability of traditional concurrency control protocols conflict with the timing constraints of real-time systems. And (2) serializability is often too strong a correctness criterion for consistency in real-time systems. Weaker consistency models are necessary to permit the application to trade off timely external consistency with strict internal consistency. Where much work in real-time transactions and databases focuses on maximizing the number and/or value of transactions which complete by deadline using traditional correctness criteria, for example [1, 8, 9, 11, 24], time-sensitive objects provide a framework for considering alternative decision rules and correctness criteria.

**Distributed Memory.** Periodic objects may be viewed alternatively as a shared memory model. Feeley and Levy [6] have argued that *versioned distributed object memory* as a programming model for distributed systems provides the benefits of simplified interprocess synchronization, increased concurrency, and latency hiding. Time-sensitive objects extend these benefits to the real-time domain. Our current model addresses only single-processor systems, but we anticipate extending it to distributed systems.

## 5 Conclusions and Ongoing Work

Time-sensitive objects are a system framework that aims to exploit the temporal characteristics of data as well as processing. The direct inclusion of validity intervals, data quality, and the extrapolation operation into the object model give the real-time application tools for maintaining forward progress in the presence of transient errors and recovering operational status in the face of more serious errors.

Periodic objects should reduce the programming complexity for real-time systems. Synchronization on versions of object value (1) hides the details of communications from both sender and receiver of the data; (2) reduces the need for direct process-to-process communication (fewer messages sent less frequently); and (3) enables clients anywhere in the system to detect missing data and infer the failure of supporting processing in a timely fashion. There is broad system support for recovery in the event of timing errors because of (1) atomicity of object operations for backward recovery to a known, consistent state, and (2) the existence of extrapolate operations for forward recovery, allowing approximation of current time-sensitive values.

There are fewer secondary factors, such as task partitioning to meet stringent deadlines, for the programmer to manage due to elimination of artificial deadline interactions. The periodic object structure itself supports direct system maintenance of time-series data. Such data is already useful

in solving a variety of real-time problems. Easy availability of such data may lead to innovative solutions to other problems.

Work is ongoing to further evaluate the feasibility of time-sensitive objects as an approach to design and implementation of real-time systems. An ideal model of time-sensitive objects has been developed to characterize the desired behavior of TSO's in the context of an environment that is not uniformly periodic. This model, which describes the desired results of interactions between objects, is being refined to accommodate the limitations of the physical world: computation time, communication time, processing and communication failures, operation conflicts. The intent of this refinement is to (1) formalize a description of expected object and system behavior; (2) formalize concepts of consistency appropriate for real-time applications, and (3) identify limitations which must be imposed on objects and their interactions to make implementation feasible. We have recently developed correctness criteria for the concurrent execution of asynchronous and periodic updates in a system of time-sensitive objects. These correctness criteria build on the work of Pu and Leff on epsilon-serializability [22] and Kuo and Mok's application of this work to real-time systems [15].

A simulation of the model is being built in parallel with model development. The simulation system is intended as a testbed for evaluating the completeness and effectiveness of the model by demonstrating the capability to describe and simulate significant real-time systems. In the future this simulation system may evolve into a prototype to be used to examine performance feasibility issues. A key issue is assessment of the costs in time and space of the recoverability features of time-sensitive objects.

## References


[1] Abbott, Robert, and Hector Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. of the 14th Int. Conf. on Very Large Databases*, Morgan Kaufmann, Palo Alto, CA, 1988, pp. 1-12.

[2] Bihari, Thomas, Prabha Gopinath, and Karsten Schwan, "Object-Oriented Design of Real-Time Software," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1989, pp. 194-201.

[3] Callison, H. Rebecca, "Time-Sensitive Objects: A Data-Oriented Approach to Real-time Systems Design," TR 91-08-11, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, August, 1991.

[4] Callison, H. Rebecca, "A Semantics for the Ideal Behavior of Time Sensitive Objects," in preparation.

[5] Chodrow, S.E., F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1990, pp 74-83.

[6] Feeley, Michael J., and Henry M. Levy, "Distributed Shared Memory with Versioned Objects," *ACM SIGPLAN Notices*, Vol. 27, No. 10 (*Proceedings of OOPSLA '92*, Oct. 1992), pp. 247-262.

[7] Gheith, Ahmed, and Karsten Schwan, "CHAOS$^{art}$: Support for Real-Time Atomic Transactions," *Proc. of the Fault Tolerant Computing Symposium*, 1989, pp. 462-469.

[8] Haritsa, Jayant R., Michael J. Carey, and Miron Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1990, pp. 94-103.

[9] Haritsa, J.R., M. Livny, and M.J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proc. of the Real-Time Systems Symp.*, IEEE Comp. Soc. Press, 1991, pp. 232-242.

[10] Herlihy, Maurice P., and Jeannette M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3 (July 1990), pp. 463-492.

[11] Huang, J., J.A. Stankovic, K. Ramaritham, and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *Proc. of the Real-Time Systems Symp.*, IEEE Comp. Soc. Press, 1991, pp. 210-221.

[12] Ishikawa, Yutaka, Hideyuki Tokuda, and Clifford W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," *SIGPLAN Notices*, Vol. 25, No. 10 (*Proc. of OOPSLA ECOOP '90*, Oct. 1990), pp. 289-98.

[13] Kenny, K.B., and K.-J. Lin, "Structuring large real-time systems with performance polymorphism," *Proc. of the Real-Time Systems Symposium*, IEEE Computer Society Press, 1990, pp. 238-46.

[14] Kopetz, Hermann, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, Vol. 9, No. 1 (February 1989), pp. 25-40.

[15] Kuo, T.-W., and A.K. Mok, "Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1992, pp.35-45.

[16] Lee, Insup, and Vijay Gehlot, "Language Constructs for Distributed Real-Time Programming," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1985, pp. 57-66.

[17] Lin, K.-J., and S. Natarajan, "Expressing and Maintaining Timing Constraints in Flex," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1988, pp. 96-105.

[18] Lin, K.-J., S. Natarajan, and J.W.S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1987, pp. 210-217.

[19] Lin, Yi, and Sang H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc.of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1990, pp. 104-112.

[20] Liu, J.W.S., K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, Vol. 24, No. 5 (May 1991), pp. 58-68.

[21] Liu, L.Y., and R.K. Shyamasunder, "Exception Handling in RT-CDL," *Computer Languages*, Vol. 15, No. 3 (1990), pp. 177-192

[22] Pu, C. and A. Leff, "Epsilon-Serializability," TR CUCS-054-90, Dept. of Computer Science, Columbia University, 1991.

[23] Raju, S.C.V., R. Rajkumar, and F. Jahanian, "Monitoring Timing Constraints in Distributed Real-Time Systems, *Proc. of the Real-Time Systems Symposium*, IEEE Comp. Soc. Press, 1992, 57-67.

[24] Sha, Lui, Ragunathan Rajkumar, and John P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *SIGMOD RECORD*, Vol. 17, No.1 (March 1988), pp. 82-98.