Object-based Semantic Real-time Concurrency Control*

Lisa B. Cingiser DiPippo and Victor Fay Wolfe
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
lastname@cs.uri.edu

Abstract

This paper presents a technique that is capable of supporting two major requirements for concurrency control in real-time databases: data temporal consistency, and data logical consistency, as well as trade-offs between these requirements. Our technique is based upon a real-time object-oriented database model in which each object has its own unique compatibility function *that expresses the conditional compatibility of any two potential concurrent operations on the object. The conditions use the semantics of the object, such as allowable imprecision, along with current system state, such as time and the active operations on the object. Our concurrency control technique enforces the allowable concurrency expressed by the compatibility function by using* semantic locking *controlled by each individual object. The real-time object-oriented database model and process of evaluating the compatibility function to grant semantic locks are described.*

1 Introduction

Many real-time control systems require the support of a *real-time database system* to manage large volumes of time-constrained data operated on by time-constrained transactions. Typical database management systems provide concurrency control techniques that seek to preserve logical consistency of data items (*e.g.* read/write locking) and logical consistency of transactions (*e.g.* two-phase locking). Although these two requirements still exist for concurrency control techniques in real-time databases, two additional requirements are also imposed: the concurrency control must support the real-time scheduling algorithm in enforcing transaction timing constraints, and the concurrency control technique must support preserving data temporal consistency constraints, which constrain data to be valid only for a certain interval of time. We are designing an object-based semantic concurrency control technique that integrates support for the four real-time concurrency control (RTCC) requirements: transaction temporal consistency, transaction logical consistency, data temporal consistency, and data logical consistency. In this paper we present the aspects of the technique that handle the data consistency requirements.

There have been many concurrency control and scheduling techniques developed to support each of the four RTCC requirements. More specifically, each of these techniques supports a *correctness criteria* that defines correct behavior of the system for one or more of the RTCC requirements. A sampling of popular correctness criteria is depicted in Table 1.

Transaction temporal consistency requirements constrain when each transaction must execute. As Table 1 shows, correctness can range from requiring no support to requiring full capability of enforcing timing constraints such as start times, deadlines, and periods on transactions.

Transaction logical consistency requirements express how a transaction must be scheduled with respect to accessing data objects. Some systems provide no support for this, while others enforce serial transactions by locking all data objects for the duration of each transaction. Less restrictive correctness criteria require serializability through techniques such as two-phase locking and timestamping. Transaction-based semantic correctness criteria allow the designer to express logical correctness of a transaction schedule based on knowledge of the specific application and of the actions performed by each transaction. These criteria are enforced through techniques such as those that employ user-defined compatibility sets of trans-

* This work has been sponsored by: The Naval Undersea Warfare Center, The National Science Foundation, and The University of Rhode Island Council For Research.

	Temporal Consistency	Logical Consistency
Transaction	• No support • Delay only required • Periodic only required • Start, deadline, period req.	• Serial Transactions • Serializable transactions • Trans.-based semantic correctness • Epsilon-serializable transactions
Data	• No support • Absolute required • Relative & absolute req.	• Mutual exclusion • Serializable operations • Object-based semantic correctness • Similarity-based correctness

Table 1: Some Correctness Criteria for the Four RTCC Requirements

actions [1]. Epsilon-serializability provides increased concurrency by allowing each transaction to define its own limits on the amount of inconsistency that it may view and that it may write [2].

Data temporal consistency correctness criteria can range from no temporal constraints to requiring absolute temporal consistency and/or relative temporal consistency of data. *Absolute temporal consistency* is maintained only if the data's value is updated within a specified time interval (so as to "accurately" reflect the environment it represents). *Relative temporal consistency* is maintained only if several specified data items have their values updated within a specified time interval (so as to have them all represent the "same" state of the environment).

Data logical consistency is the type of consistency (data integrity) that is maintained in a traditional database. One form of data logical consistency requires mutual exclusion, which is supported using techniques such as exclusive locking and monitors. Another form requires serializable execution of operations on the data; it is supported by techniques such as read/write locking, and commutativity-based scheduling [3]. Object-based semantic logical correctness, which is based on the designer's knowledge of how the data is used, is supported by semantic locking techniques [4, 5] in which the user defines compatibility of locks on the data item. Several correctness criteria have been recently introduced for *similarity-based* scheduling which define correct schedules based on the amount of imprecision introduced into the data [6].

Unfortunately, techniques to support one form of RTCC requirement are often not well-suited to support another. For instance, many traditional concurrency control techniques support only serial or serializable transaction schedules. There are two major problems with the use of these techniques in real-time databases. First, they are only concerned with preserving typical logical correctness of data and transactions; they are not concerned with temporal consistency of data or transactions. Second, allowing only serial or serializable schedules is too restrictive for real-time databases that must schedule transactions to meet timing constraints. As Figure 1 depicts, serial schedules are a small subset of all possible logically consistent schedules. Serializable schedules are typically a bigger, but still small, subset of all logically consistent schedules. A larger subset of logically consistent schedules is made up of those schedules which may be expressed as semantically correct. While it is true that, in theory, all logically consistent schedules are semantically correct, the set in Figure 1 is depicted as a subset because, in reality, not all logically consistent schedules can be easily expressed using the semantics of the application. Allowing more logically consistent schedules is important in a real-time database: the more logically consistent schedules that the concurrency control allows, the more flexibility the database manager's scheduler has in determining a logically consistent schedule that

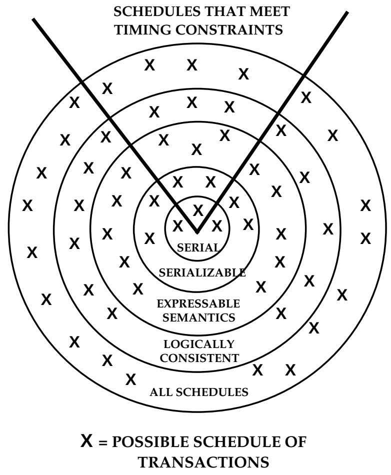

Figure 1: Concurrency Control Allowable Schedules

meets timing constraints. There has been significant work on scheduling real-time transactions [7, 8]; however, no widely applicable optimal scheduling algorithm has been found that will generate a logically consistent schedule that meets timing constraints if such a schedule exists. Thus, a real-time concurrency control technique should allow as many logically consistent schedules as possible to provide the database management system scheduler with flexibility in determining a schedule that meets all constraints.

Designing a concurrency control technique that meets all four forms of RTCC requirements is difficult because these requirements can have general fundamental conflicts. For instance, preserving temporal consistency of a data item may require preempting a transaction that is using the data item in favor of an update transaction. However, this preemption may violate the logical consistency of the data item and/or the logical consistency of the preempted transaction. Therefore, a concurrency control technique that seeks to support all four forms of RTCC requirements must be capable of expressing the tradeoffs in sacrificing one RTCC requirement for another. These tradeoffs are application-specific; that is, an application may favor one RTCC requirement over another depending on the circumstances. The decision regarding which RTCC requirement to enforce is based on the system conditions and on the application semantics.

To integrate support for the RTCC requirements and the tradeoffs that they introduce, we have developed an object-based *semantic real-time concurrency control* technique. Object-based semantic concurrency control techniques allow the user to specify allowable interleavings of object operations to preserve data logical consistency; some of these are presented in [3, 9, 5, 4]. However, these techniques typically only support limited forms of logical consistency; they do not address temporal consistency or the tradeoffs among the RTCC requirements.

In Section 2 we present a portion of a general model of an object-oriented real-time database system that we use to describe our concurrency control technique. Section 3 describes our technique, which uses *semantic locking* of data objects, where compatibility of locks is based on current system conditions, including timing and allowable *imprecision* of data and transactions. While it is important to maintain all four RTCC requirements, this paper concentrates on how to handle data consistency requirements (temporal and logical) and makes certain assumptions about correctness criteria upheld by transactions. Also note that our technique is designed for soft real-time systems: it makes a best effort to meet timing constraints, but offers no guarantees. Section 4 summarizes and discusses how our technique supports the data object RTCC requirements and the tradeoffs between them.

2 Real-Time Object-Oriented Model

Most work in developing concurrency control for real-time database management systems uses the relational data model [10]. Although the relational model is useful for many applications, there are several reasons why we believe that it is not as well-suited as an *object-oriented database model* (OODM) (for a survey of object-oriented database research see [11]) for supporting semantic real-time concurrency control. First, the encapsulation mechanisms of an OODM allow concurrency control specific to a data object to be enforced within the object. That is, instead of imposing a general correctness criteria, such as serializability, on all data objects, the allowable concurrency of an object can be treated as a component specific to an object or class of objects. Second, the capability to include user-defined operations (methods) on data objects can improve real-time concurrency by allowing a wide range of operation granularities for semantic real-time concurrency control. That is, instead of only enforcing concurrency among read and write operations, as is typically done in relational data models, the OODM can potentially allow for enforcing concurrency among the rich set of user-defined operations on objects. Finally, an OODM potentially makes it easier to integrate constraint expression and checking as compared to relational models [11]. This improved constraint handling is particularly important when we consider the constraints imposed by the RTCC requirements. We exploit these advantages in the model of an object-oriented real-time database that we present in this section.

2.1 Model

We model a real-time database as a *database manager*, a set of *objects*, a set of *relationships* and a set of *transactions*. The database manager performs typical database management operations including scheduling of all execution on the processor, but not necessarily including concurrency control (in the technique described in Section 3, objects and transactions coordinate to perform concurrency control themselves). We assume that the database manager uses some form of real-time, priority-based, preemptive scheduling of execution on the processor[1]. Database *Objects* represent database entities. *Relationships* are objects that represent associations among the database objects. *Transactions* are executable entities which access the objects and relationships in the database.

Objects. An *object* is defined by $\langle N, A, M, C, CF \rangle$. The component N is a unique name or identifier for the object. The component A is set of attributes, each of which is characterized by $\langle V, T, I \rangle$. V is a complex data type that represents some characteristic value of the object, T is a time field which defines the age of the attribute, and I is a boolean imprecision field that identifies whether or not some imprecision has been introduced into the value of the attribute.

An object's M component is a set of methods which are the only means transactions have of accessing the attributes in the object. A method is defined by $\langle O, exec, TS \rangle$. O is a sequence of programming language statements including: conditional branching, looping, I/O, and reads and writes to the object's attributes. Read and write operations on attributes read and write the time and imprecision fields, as well as the value field. *Exec* is the worst-case execution time of the method. TS is a set of *temporal scopes*, each of which defines absolute timing constraints on part of the method's execution and is represented by $\langle E, sa, sb, d \rangle$. E is the subsequence of O that is to be time constrained, sa is an absolute earliest start time, sb is an absolute latest start time and d is an absolute latest complete time (deadline). A temporal scope expresses the constraint: $\forall_{e \in E}((sa \leq start(e) \leq sb) \land (complete(e) \leq d))$.

The C component of an object is a set of constraints which define the correctness of the object with respect to the system specification. A constraint is defined by $\langle Pr, ER \rangle$. Pr is a predicate represented in a boolean algebra with special atoms defined to signify the changing of an attribute ($change(a)$), the time at which an attribute becomes temporally inconsistent ($deadline(a)$), the allowable amount of imprecision for an attribute ($ImpAmt(a)$), the start time of an execution ($start(e)$), and the completion time of an execution ($complete(e)$). An execution e is a single executable entity such as a method invocation, or a simple read operation. *Logical constraints*, such as integrity and range constraints, are specified using the V fields of attributes. *Temporal consistency constraints* are specified using the T fields of attributes. Imprecision can be bounded by using constraints that involve attribute's I field and the $ImpAmt$ characteristic.

Violating a constraint amounts to making the constraint's predicate false. The ER component of a constraint is an *enforcement rule* which is a sequence of programming language statements including reads and writes on the object's attributes that is executed when the constraint is violated.

The CF component of an object is a *compatibility function* with domain $M \times M$ that expresses conditional compatibility between all pairs of methods in the object. We describe the CF component in detail in Section 3.

Other Model Components. A *relationship* is a special kind of object that is used to express associations among other objects. It has all of the components of an object, and it can also express inter-object constraints among the participating objects. A *transaction* accesses objects by invoking their methods. It can define a set of method invocations to be atomic and/or exclusive of incompatible interruption. Transactions have timing constraints and may express requirements for temporally and/or logically consistent data. A complete formal description of the entire model, called the RTSORAC (**R**eal-**T**ime **S**emantic **O**bjects **R**elationships **A**nd **C**onstraints) model, including detail on relationships, transcations, and inheritance, can be found in [12].

2.2 Example

We illustrate our object-oriented real-time database model using an application in submarine command and control systems that involves contact tracking, contact classification, and response planning tasks that must have fast access to large amounts of sensor data [13]. Since sensor data is only valid for a certain amount of time, the database system must ensure the temporal consistency of the data so that transactions, such as those for contact tracking and for response planning, get valid data. All data in the system may have to be accessed by transactions that have timing constraints, such as those involved with tracking other ships in a combat scenario.

Figure 2 illustrates an example of a **Submarine** object type in the database schema. Recall that along with each attribute value is the time field representing the age of the value. In some cases, as with the static *Size* attribute, this time will be ∞ since it has no real-time characteristics. Other attributes, such as *Speed* and *Bearing*, will be updated periodically. To specify the temporal consistency of these attributes, timing

[1]We assume a single processor system for simplicity, our technique can be extended to multi-processor systems.

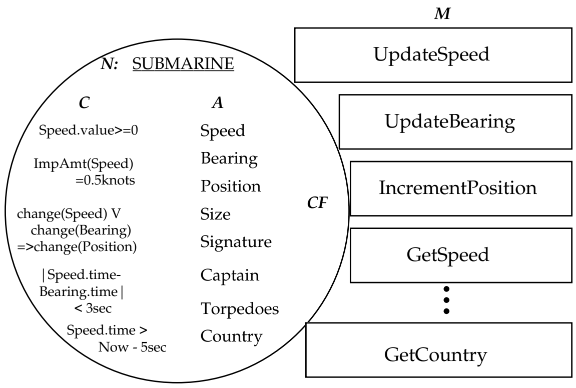

Figure 2: Example of **Submarine** Object Model

constraints are placed on the object indicating when the attributes must be updated. For example, the absolute temporal consistency constraint on the *Speed* attribute has a predicate that states $Speed.time > Now - 5sec$ and the corresponding enforcement rule marks the *Speed* as invalid if the constraint is violated. The relative temporal consistency constraint predicate $|Speed.time - Bearing.time| < 3sec$ expresses that the time fields of the *Speed* and *Bearing* attributes must be within three seconds of each other.

The example imprecision constraint:

$$ImpAmt(Speed) = 0.5knots$$

specifies that data may be written that introduces no more than $0.5knots$ of imprecision to the *Speed* (possibly due to interleavings allowed by the concurrency control as we describe in Section 3). If the *imprecision* field's value is TRUE, then some imprecision has already been introduced into the *Speed* attribute; this information will assist in bounding the amount of imprecision allowed in the object.

3 Semantic Real-Time Concurrency Control Technique

Our approach to supporting the RTCC requirements is to provide a semantic concurrency control technique in which each object has a concurrency control (CC) mechanism that uses its (user-defined) compatibility function to grant semantic locks to transactions. This section describes the compatibility function and many of the important system characteristics it may capture in its conditional expression of compatibility. The semantic locks provide transactions with permission to invoke specified methods of the object. Since this paper concentrates on the object-based concurrency control, we assume that transactions use a two-phase locking scheme to coordinate transaction logical correctness[2]. This section describes the semantic locks and the process that an object performs when a transaction requests one.

3.1 Compatibility Function

To express allowable concurrency of method invocations, we expand on previous object-based semantic concurrency control techniques that define a compatibility table for an object [4, 5]. In these techniques, a pair of method invocations is compatible, and hence the entry in the table is TRUE, if the invocations can always be executed concurrently and still preserve the logical consistency of the data object. The compatibility function of an object extends this notion to express compatibility as a run-time function of the form:

$$CF(M_a(a_1, a_2, \ldots, a_n), M_r(b_1, b_2, \ldots, b_m)) = \langle BooleanExpression \rangle$$

Here, M_a is a currently active method invocation with arguments $a_1 \ldots a_n$, and M_r is a method invocation (with arguments $b_1 \ldots b_m$) that has been requested by a transaction. The boolean expression may involve predicates for any of the current system characteristics that we describe below.

Affected Set System Characteristics. The potential increase in concurrency provided by semantic concurrency control is particularly important in object-oriented databases where data objects can be large and complex so that locking an entire data item is often unnecessary and inefficient. Instead, the semantics of the object methods can be used so that if two method invocations do not affect the same part of the object, they can potentially execute concurrently without violating the data logical consistency constraints of the object. In order to determine what data is affected by a particular method invocation, we modify the concept of an affected set originally presented in [3]. When a method is invoked, the set of attributes affected by the particular invocation is computed. The affected set must be computed for each invocation of a method because the affected attributes may depend on the arguments of the method. The attributes in the affected set for a method include all of the attributes that are read by or written by the method as well as any attributes affected by enforcement rules that may be triggered by the method. We can also refer to subsets of the affected set that

[2]We assume two-phase locking for simplicity, our technique can be extended to other techniques of preserving transaction logical correctness.

A: $CF(IncPos(Amt_1),\ IncPos(Amt_2)) =$
$(Amt_1 < ImpAmt(Position))\ AND$
$(Amt_2 < ImpAmt(Position))\ AND$
$(NOT(Position.imprecise))$

B: $CF(GetSpeed(S_1), UpdateSpeed(S_2)) =$
$(Speed.time < Now - 5)\ AND$
$(|Speed - S_2| < ImpAmt(Speed))\ AND$
$(GetSpeed.temporal)\ AND$
$(NOT(GetSpeed.logical))$

C: $CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2)) =$
$(|S_1 - S_2| < ImpAmt(Speed))\ AND$
$(NOT(Speed.imprecise))$

Figure 3: Compatibility Function Examples

contain only attributes read by the method (*read affected set*) or only attributes written by the method (*write affected set*). The affected set can be used to define compatibility between methods. For instance, in [3] two methods are considered to be compatible if the intersection of their affected sets is empty. We represent the affected sets of method m_i in the compatibility function as characteristics: $Affected(m_i)$, $ReadAffected(m_i)$, $WriteAffected(m_i)$, with which we can form predicates by testing for membership and empty intersections.

Active Method Invocations System Characteristics. *Active method invocations* are method invocations of an object that the object's CC mechanism currently allows to execute. The set of active method invocations for an object is represented by a set called *Active* that contains all active method invocations of that object and by a set of counters, $Active(m_i)$, each of which contains the number of active invocations of its corresponding method m_i. We use the active method invocations characteristics to mitigate a drawback of object-based semantic concurrency control techniques that use compatibility tables: that compatibility tables such as [4, 5] represent strictly binary relations between methods. This drawback implies that there is no way of expressing a maximum number of a particular method invocation that can be concurrently active or that two method invocations are compatible only if a third method invocation is not already active. For instance, a compatibility table can not express that an object which represents a video image may allow two method invocations that write to the image (and cause temporary blurring) to execute concurrently as long as no method that views the image is currently active.

Time-based System Characteristics. Time may be used in a compatibility function to facilitate maintaining temporal consistency of the object. The compatibility function's boolean expression can include predicates involving time fields of attributes (represented by *.time*), time values such as the current time (Now), and the time at which an attribute a becomes temporally inconsistent ($deadline(a)$). As an example of how time-based system characteristics can be used to express a particular object's requirement that temporal consistency should be enforced at the expense of logical consistency, consider the condition expressed in Figure 3B. In this example, a method invocation of *GetSpeed* is currently active and a method invocation of *UpdateSpeed* has been requested. Normally, these methods would be incompatible because *UpdateSpeed* might cause the data being read by *GetSpeed* to be changed. However, in this example the semantics of the application determined that if the temporal consistency constraint requiring that the *Speed* attribute be no more than five seconds old is violated, then *UpdateSpeed* is allowed to execute so that a new value for *Speed* can be written to restore temporal consistency.

Imprecision System Characteristics. Imprecision is used to provide added flexibility in scheduling, as well as to allow the tradeoff of logical consistency for temporal consistency. That is, certain interleavings that may introduce an allowable amount of logical imprecision to attribute values might be acceptable if they help to maintain temporal consistency or if they provide more scheduling flexibility. For instance, consider the increment position method invocation, *IncPos*, in the example of Figure 3A that performs a read ($r(Position)$) operation followed by a write ($w(Position + Amt)$) operation. The possible outcomes of interleaving two invocations of *IncPos* are either that both increments affect the *Position*, as with a serial schedule, or only one of them does, as in the following schedule in which only Amt_1 is added to the *Position*:

$$r_1(Position), r_2(Position), w_2(Position + Amt_2), w_1(Position + Amt_1).$$

If both *Amt* arguments are less than the allowable amount of imprecision for *Position*, then the result of any interleaving will be within the allowable bounds of imprecision and the two method invocations can be allowed to be concurrent; otherwise they will not be allowed to be concurrent. Another example of allowing imprecision is shown in Figure 3B, where the requested *GetSpeed* method invocation is compatible with the active *UpdateSpeed* invocation only if the value being written to the *Speed* attribute is within the imprecision bounds for the attribute.

For each attribute in an object, there can be a specification of how much imprecision is allowed, defined by the *ImpAmt* characteristic. If *ImpAmt* is unspecified, the attribute must be precise. The compatibility function should specify interleavings that introduce at most the defined amount of imprecision. However, if the CC mechanism allows the same interleaving more than once, it may introduce more imprecision than the constraint allows. The imprecision field of each attribute (represented by *.imprecise*), along with the *ImpAmt* constraint can be used in the compatibility function to bound imprecision of each attribute. For instance, the compatibility function may have a clause that checks the value of the *.imprecise* field of an attribute and if the value is TRUE, the CC mechanism disallows any further interleavings that could potentially introduce imprecision. Such a clause is captured in Figure 3A: if the *Position* attribute is marked as imprecise, then the interleaving is not allowed. Note that there is no way of knowing exactly how much imprecision has been introduced into an attribute because that depends upon the exact interleaving of method invocations that is scheduled. The semantics of each possible pairing of method invocations determines the maximum amount of imprecision that could be introduced if they were allowed to execute concurrently. We describe in Section 3.3 how each attribute's *.imprecise* field is set.

Method Arguments System Characteristics. Since the way in which method invocations affect an object often depends on the arguments of the method invocation, it may be important to examine the arguments of the method invocations when determining compatibility [4]. For instance, if two method invocations write to the same attribute, then they may be allowed to interleave if they both write the same value. Another example is shown in Figure 3A: when determining the compatibility between two invocations of the method *IncPos*, the methods' arguments are examined to determine if they are both within the allowed amount of imprecision for the *Position* attribute.

Invoking Transactions System Characteristics. The consistency of data required by the invoking transactions (see Section 2) can also affect whether a method invocation should become active. In order to specify its consistency requirements, a transaction defines each of its method invocations to require temporal and/or logical consistency. These specifications can then be used in clauses of the compatibility function, represented by boolean fields *.temporal* and *.logical*. In the example of Figure 3B, the compatibility function's clause *GetSpeed.temporal* only allows method invocation *UpdateSpeed* to interleave with method invocation *GetSpeed* if the transaction invoking *GetSpeed* requires temporal consistency. Otherwise, *GetSpeed* will not be interrupted by *UpdateSpeed*, even if a temporal consistency would be violated.

3.2 Semantic Locks

In our semantic real-time concurrency control technique, the CC mechanism of each object uses *semantic locking* to enforce the allowable concurrency expressed by the compatibility function of the object. A transaction must acquire a semantic lock for a method invocation before the method is allowed to execute. When a transaction requests a semantic lock from an object, the request contains the identifier for the requesting transaction, an indicator of the transaction's precision requirements, the method to be locked and the arguments to the method if they are available. A transaction may request a group of semantic locks for a set of future method invocations in order to enforce its exclusive constraints. For instance, a transaction that wishes to perform several method invocations, $mi_1 \ldots mi_n$, exclusively on an object (without interference from incompatible method invocations) would request semantic locks for all of the method invocations in $\{mi_1 \ldots mi_n\}$. The transaction would only invoke the methods once it had all of the semantic locks and would only release the locks once all of $mi_1 \ldots mi_n$ had completed. Note that since we assume for this paper that transactions perform two-phase locking, it may be necessary for a transaction that invokes more than one method to request all semantic locks prior to the method invocations.

3.3 Semantic Locking Process

When a transaction requests a semantic lock from an object, there are two possible outcomes: either the lock is granted immediately, or the request is placed on a priority queue within the object to be granted later. The specific outcome for each lock request is based on evaluating the compatibility function of the object. That is, the conditions expressed in the compatibility function determine whether locks are granted. Most other locking techniques require evaluating a condition for concurrency control (*e.g.* typically a read lock is granted only under the condition that a write lock is not currently active), however our technique employs user-defined conditions based on the semantics of the application.

Requesting Semantic Locks. Figure 4A depicts the process conducted by an object's CC mechanism in response to a semantic lock request. First, the CC mechanism performs a "Check Compatibilities" function that uses the current lock request and evaluates the compatibility function once for every active lock and for every queued request of higher priority. If the current request is incompatible with any active lock or any queued request of higher priority, it is placed in the priority queue according to the priority of the requesting transaction. Otherwise, the semantic lock is granted to the transaction and the lock is added to the active locks set for the object. If a semantic lock is requested before its associated method is invoked, the values of the method's arguments are not available at the time the lock is requested. Therefore, any predicate of the compatibility function involving method arguments of such a semantic lock evaluates to FALSE. For example, in Figure 3C, assume that a semantic lock is held for an anticipated method invocation $UpdateSpeed_1$ and a semantic lock is requested for another invocation, $UpdateSpeed_2$. Since argument S_1 is not known, the lock on $UpdateSpeed_2$ is considered incompatible with the lock on $UpdateSpeed_1$.

Checking compatibilities also has the side effect of setting the imprecision fields of attributes. For each attribute a in the write affected set of the current lock request, if any evaluation of the compatibility function for the current request requires a check of the imprecise field of a, then $a.imprecise$ is set to TRUE. This side effect is performed because we assume (pessimistically) that any condition which is concerned with the imprecision of an attribute may allow concurrency that could introduce imprecision into the attribute. If none of the evaluations of the compatibility function checks $a.imprecise$, then the execution of the current request is assumed to introduce no imprecision to a and, if the request is granted a lock, $a.imprecise$ is set to FALSE. Note that since the imprecision field is set as compatibilities are checked, only one active method can introduce potential imprecision to attribute a. However, if $a.imprecise$ is set to TRUE by the check of compatibilities and the lock is eventually not granted, $a.imprecise$ is reset to FALSE.

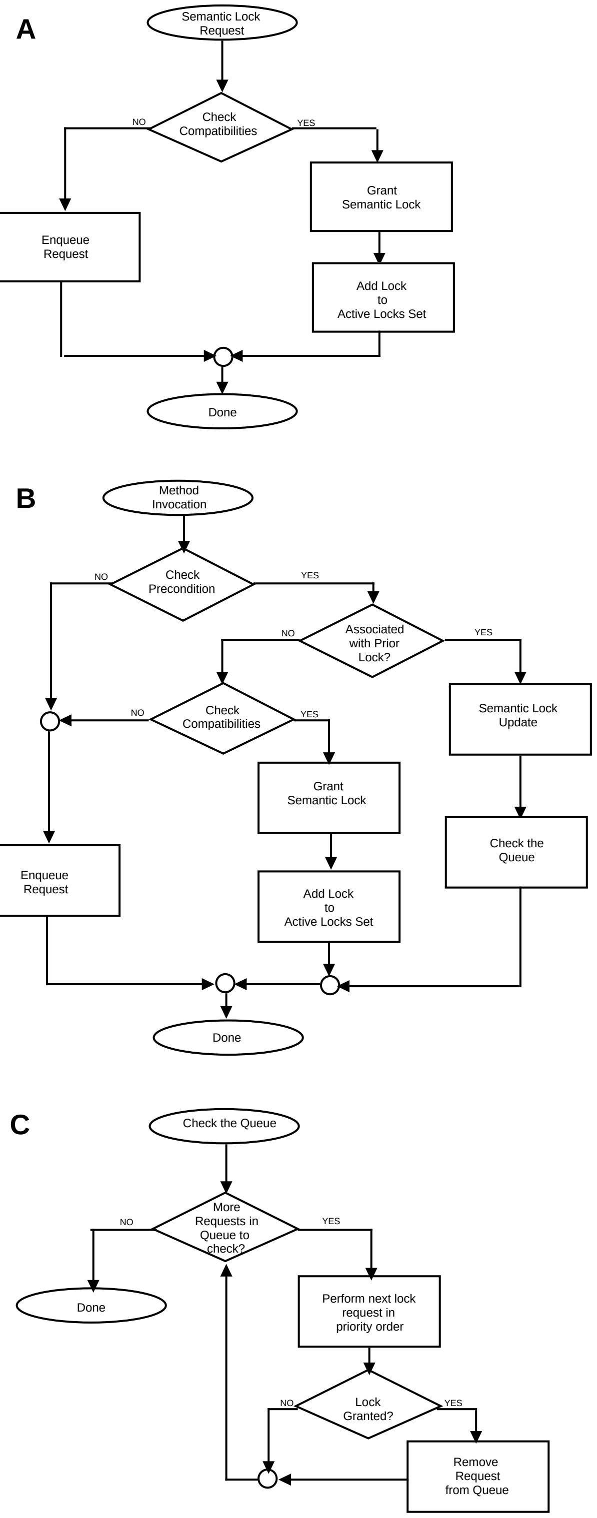

Figure 4: Process For Requesting Locks

Method Invocation. Figure 4B depicts the process employed by an object's CC mechanism in response to a method invocation. First, the mechanism tests a *precondition*. The precondition attempts to ensure that if the requesting transaction requires temporal consistency, then none of the attributes in the read affected set of the invoked method will become temporally inconsistent while the method is executing. The precondition examines the deadline of every read affected attribute to determine if the method's execution time is longer than any of the attribute deadlines. That is, an invocation of a method m with execution time $exec(m)$, requires that the following precondition holds: $m.temporal \Rightarrow \forall_{a \in ReadAffected(m)}$ $(exec(m) < deadline(a) - Now)$. Notice that if temporal consistency is not required by the invoking transaction, the precondition is always met. If the precondition fails, then executing the method would ensure that the transaction gets temporally inconsistent data. Therefore the method invocation is enqueued until the precondition can be met (usually after an update to the temporally inconsistent attribute).

If the precondition is met, then the object's CC mechanism looks for a previously obtained semantic lock for the method invocation. If no lock was requested earlier, a lock must be obtained at this point, so the CC mechanism checks compatibilities with the current method invocations. If any incompatibilities arise, the request is enqueued. Otherwise, the semantic lock is granted and it is added to the active lock set.

If the invoked method is associated with a previously obtained semantic lock, the object's CC mechanism performs a *Semantic Lock Update*. In this procedure, any of the information that the semantic lock lacked at the time of its request, such as specific method arguments, is copied from the method invocation into its semantic lock. Because some of the requests waiting in the queue may now be compatible with this semantic lock, the queue is checked for any newly compatible requests (see Figure 4C).

Releasing Locks. A semantic lock may either be explicitly released by request of the holding transaction or implicitly released upon completion of method execution or when a transaction commits or aborts. Whenever a semantic lock is released, the priority queue is checked for any requests which may be granted (see Figure 4C). Since the newly released semantic lock may have been on a method that restored temporal consistency to an attribute, or may have been a semantic lock that had caused some other incompatibilities, some queued requests may now be granted locks. The requests in the wait queue are reissued in priority order, and if any of these requests is granted, it is removed from the queue.

4 Conclusion

This paper has described an object-based real-time semantic concurrency control technique that can support both the logical and temporal data consistency RTCC requirements of a real-time database as well as the tradeoffs between them.

The time field of each data attribute and the explicit constraints of each object allow absolute temporal consistency (using the time field and an absolute time) and relative temporal consistency (using the time fields of multiple attributes) to be specified. These explicit temporal constraints can be captured in clauses of the compatibility function so that our semantic real-time concurrency control technique can determine if data is temporally consistent when deciding on allowable concurrency. The technique does this determination in two ways. First, it tests the precondition (see Section 3.3) and only allows a method invocation that requires temporally consistent data to be executed if it appears that the data will be temporally consistent while the transaction uses it. Otherwise, the transaction is queued until the data's temporal consistency is restored. Second, the compatibility function allows general specification of temporal consistency conditions so that concurrency can be controlled based on the temporal consistency status of any attribute in the object. In addition, our technique actively supports the maintenance of data temporal consistency by allowing conditions to specify that update methods can be executed to restore temporal consistency at times where typical locking techniques might disallow that execution.

Various data logical consistency correctness criteria can be maintained by our technique. For example, exclusive locking, which allows only serial object operations, can be realized by making every pair of method invocations incompatible. Serializability of object operations can also be maintained by building the traditional read/write locking compatibilities into the compatibility function. Object-based semantic correctness is facilitated by the expressive power of the compatibility function. In fact, our semantics can go beyond semantic specification of precise correctness by including a specification for an allowable amount of imprecision in the data.

Note that in this paper we have only described the concurrency control mechanism within a single object. In a database system based on the RTSORAC model (see Section 2 and [12]), multiple objects can be connected by relationship objects that express inter-object constraints. We are developing a technique in which semantic locks may be propagated from object to object via relationship objects in order to enforce inter-object constraints.

Our technique also supports the inevitable tradeoffs among the RTCC requirements by allowing the specification of relaxing one form of constraint in favor of another in certain circumstances. We demonstrated how the compatibility function can specify that logical consistency constraints can be relaxed, either completely or by allowing bounded imprecision, in order to preserve temporal consistency. Furthermore, transactions can specify whether they require temporal consistency or precise logical consistency so that this information can be used in determining the tradeoff.

The power and flexibility of our approach can also be its drawback. Specifying such a flexible system is a significant responsibility. The modularity of using object-based semantics provides some relief since one can focus on the semantics of each individual object, which is more tractable than having to consider the semantics of all objects and all transactions. We are developing software tools that further ease the burden on the object designer by computing a default version of the compatibility function for each object and then allowing the object designer to interactively adjust the conditions to include the specifics of the application semantics. However, the fact remains that maintaining correctness in a real-time database is more complex than in a traditional database system due to temporal consistency requirements and allowed imprecision. We believe that this complexity requires flexible, modular concurrency control, such as the object-based semantic technique that we have described here.

Acknowledgements. We thank Joan Peckham, Janet Prichard, Krithi Ramamritham, Jack Stankovic, Jim Oblinger, and Azer Bestravos for their helpful comments, suggestions, and support.

References

[1] 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.

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

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

[4] P. M. Schwartz and A. Z. Spector, "Synchronizing shared abstract types," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 223–250, 1984.

[5] V. F. Wolfe, S. Davidson, and I. Lee, "*RTC*: language support for real-time concurrency," *Real-Time Systems.* vol. 5, no. 1, pp. 63-87, March, 1993.

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

[7] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," In *Proceedings of the 14th VLDB Conference*, August 1988.

[8] J. Haritsa, M. Livny, and M. Carey. "Earliest deadline scheduling for real-time database systems," In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1990.

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

[10] K. Ramamritham, "Real-time databases,". *International Journal of Distributed and Parallel Databases*, vol. 1, no. 2, 1993.

[11] S. Zdonik and D. Maier, *Readings in Object Oriented Database Systems.* San Mateo, CA: Morgan Kauffman, 1990.

[12] V. F. Wolfe, J. Peckham, L. Cingiser (DiPippo), and J. Prichard, "A model for a real-time object-oriented database," Tech. Rep. TR-93-216, University of Rhode Island Dept. of Computer Science, May 1993. Presented at the *IEEE Workshop on Real-Time Operating Systems and Software*, March, 1993.

[13] G. A. Bussier, J. Oblinger, and V. F. Wolfe, "Real-time considerations in submarine target motion analysis," *Proceedings of the First IEEE Workshop on Real-Time Applications*, March, 1993.