Object-based Semantic Real-time Concurrency Control With Bounded Imprecision*

Lisa Cingiser DiPippo and Victor Fay Wolfe
Department of Computer Science
University of Rhode Island
Kingston, RI 02881

Abstract

This paper describes a concurrency control technique for real-time object-oriented databases that supports logical consistency and temporal consistency, as well as bounded imprecision that results from their trade-offs. The concurrency control technique uses a *semantic locking* mechanism within each object and user-defined conditional compatibility over the methods of the object. The semantics can specify when to sacrifice precise logical consistency to meet temporal consistency requirements. It can also specify accumulation and bounding of any resulting logical imprecision. We show that this technique, under certain general restrictions, can preserve global correctness and bound imprecision by proving it can guarantee a form of *epsilon serializability* specialized for object-oriented databases.

Index terms: *bounded imprecision real-time object-oriented databases, semantic concurrency control*

1 Introduction

Real-time applications such as air traffic control, autonomous vehicle control, and automated manufacturing involve large amounts of environmental sensor data. These applications are supported by *real-time database management systems* (RTDBMS) [1]. In addition to supporting typical *logical consistency* requirements, a RTDBMS concurrency control technique must maintain *temporal consistency constraints. Data temporal consistency* constrains how "old" data can be while still being considered valid. *Transaction temporal consistency* constrains when transactions can execute and be considered correct.

Traditional DBMS concurrency control techniques are designed to enforce only logical consistency constraints, but not temporal consistency constraints on data values and transaction execution. For instance, a typical serializability-based concurrency control technique might disrupt an earliest-deadline-first transaction scheduling order by blocking a transaction with a tight deadline in favor of a transaction with a looser deadline in order to maintain logical consistency by

*This work has been sponsored by the Naval Undersea Warfare Center and the National Science Foundation preserving the serialization order of transactions. Serializability-based techniques can also be a problem in a RTDBMS because they restrict allowed concurrency, often more than is required for logical correctness [2]. This over-restriction impedes a real-time transaction scheduler's ability to preserve transaction temporal consistency because requiring serializability reduces the likelihood of creating a schedule that meets timing constraints [3]. Data temporal consistency is also ignored by serializability-based concurrency control techniques. For instance, a serializability technique would block a transaction t_{update} that updates temporally inconsistent data if another transaction t_{read} is reading the data. This blocking might cause t_{read} to receive temporally inconsistent data. On the other hand, relaxing serializability by allowing transaction t_{update} to preempt transaction t_{read} could violate the logical consistency of t_{read}. As this example indicates, the requirements of meeting logical and temporal consistency constraints can conflict with each other.

There have been proposals for techniques that relax serializability [2, 4, 5, 6, 7, 8]. Many of these techniques use semantic knowledge of the system to determine logical correctness, instead of mandating a serializable schedule. However, these techniques were not intended for RTDBMSs and thus do not incorporate semantics associated with temporal consistency. A survey of real-time database concurrency control issues is presented in [9]. Many of these techniques relax serializability, but still neglect data temporal consistency considerations. The exception is work presented in [10], that replaces serializability with a correctness criterion called *similarity*. Similarity is a semantically defined relation between a pair of data values that indicates that the values are recorded "close enough" in time to be considered equal. It is used to define a concurrency control technique that incorporates temporal consistency considerations. This technique, however, does not directly address both logical consistency and temporal consistency.

We have designed a concurrency control technique [3] called *semantic locking* that supports expression and enforcement of the trade-offs between logical and temporal consistency constraints for *real-time object-oriented database management systems*. Our technique is designed for *soft real-time* data management, which means that it makes an effort to preserve temporal consistency, but can offer no *a priori* guarantee of meeting timing constraints. Due to its lack of guarantees, the technique is not appropriate for *hard real-time* data management, where timing constraints must be predictably met. In our semantic locking technique, concurrency control is distributed to the individual data objects, each of which controls concurrent access to itself based on a semantically-defined *compatibility function* for the object's methods [3]. This semantically-defined compatibility is similar to that described in [8, 11] which use the notion of commutativity to define operation conflict. The semantics allowed in our technique are richer than those allowed in [8, 11] because our semantics include, among other things, expression of conditions under which logical consistency should be relaxed in order to maintain temporal consistency. For instance, in the above example, the semantics could express that transaction t_{update} be allowed to write the data item that transaction t_{read} is reading only under the condition that temporal consistency of the data item is threatened or violated.

If a RTDBMS concurrency control technique sacrifices logical consistency to maintain temporal consistency, it may introduce a certain amount of logical *imprecision* into data and/or transactions. For instance, in the above example, if the concurrency control technique allows transaction t_{update} to write the data item while transaction t_{read} is reading the data item, then t_{read} might get an imprecise view of the data. The data item itself might become imprecise if two transactions that write to it are allowed to execute concurrently. While imprecision in a database is not desirable, it is often tolerable. For instance, in an air traffic control application, a transaction that queries for the position of all airplanes within an airspace may read-lock the position data items for a long duration in order to gather all of the data. During this transaction's execution, it could be desirable to allow updates to the read-locked data items in order to maintain their temporal consistency. Updates of read-locked data could introduce imprecision into the querying transaction's view of the positions of the tracked aircraft. However, the application may specify that it is sufficient for the values of the relative position data to be within a specified range of exact values. That is, the application may allow some bounded imprecision in the transaction's return values. However, allowing imprecision to become unbounded in the database is not acceptable.

In this paper, we describe our semantic locking technique and how it can specify accumulation and bounding of logical imprecision that results from the trade-off of logical consistency for temporal consistency. We also derive two general restrictions on the expressed semantics and show that these restrictions are sufficient for bounding logical imprecision in the system. We formally prove the sufficiency of the restrictions by demonstrating that our semantic locking concurrency control technique, under the restrictions, guarantees a form of *epsilon serializability* (ESR) [12] specialized for object-oriented databases. ESR is a formal correctness criterion that specifies that a schedule for transaction execution is correct if the results of the schedule (both data values and transaction return values) are within specified limits of a serializable schedule. By demonstrating that our technique can maintain a version of ESR, we show that it can provide logical correctness while better enforcing temporal consistency.

Section 2 presents our model of a real-time object-oriented database. Section 3 describes the semantic locking technique. Section 4 first describes the ESR correctness criteria and extends it to our model of a real-time object-oriented database. The section then presents the two general restrictions on the expressed semantics and proves that the semantic locking technique, under these restrictions, meets the object-oriented ESR correctness criteria. Section 5 summarizes and compares our work to related work.

2 RTSORAC Model

Our RTDBMS semantic locking concurrency control technique is based upon our model of a real-time object-oriented database called RTSORAC (**R**eal-**T**ime **S**emantic **O**bjects, **R**elationships **A**nd **C**onstraints) [13]. This model extends object-oriented data models by incorporating time into objects and transactions. This incorporation of time allows for explicit specification of data temporal consistency and transaction temporal consistency. The RTSORAC model is comprised of a *database manager*, a set of *object types*, a set of *relationship types* 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. We assume that the database manager uses some form of real-time, priority-based, preemptive scheduling of execution on the processor. Database *object types* specify the structure of database objects. *Relationships* are instances of relationship types; they specify associations among the database objects and define inter-object constraints within the database. *Transactions* are executable entities that access the objects and relationships in the database. This paper focuses on bounding imprecision in objects and transactions, so in presenting the RTSORAC model, we concentrate describing the model for object types and transactions. The model for relationship types is described in more detail in [13].

We illustrate our real-time object-oriented database model using a simplified submarine command and control system. The application involves contact tracking, contact classification and response planning tasks that must have fast access to large amounts of sensor data [14]. This sensor data is considered precise and thus provides a periodic source of precise data to the database. 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 response planning, get valid data. The data in the system may be accessed by transactions that have timing constraints, such as those involved with tracking other ships in a combat scenario. Transactions in this application may also allow for certain amounts of imprecision depending on

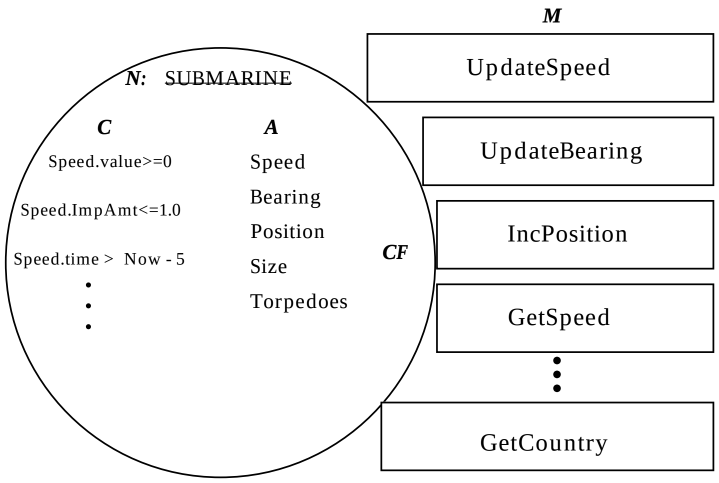

Figure 1: Example of **Submarine** Object Type

the semantics. For instance, a transaction that requests position information involving a friendly ship may allow more imprecision than a transaction tracking ships in a combat scenario. Figure 1 illustrates an example of a **Submarine** object type in the database schema.

2.1 Object Types

An *object type* is defined by $\langle N, A, M, C, CF\rangle$. The component N is the name of the object type. The component A is a set of attributes, each of which is characterized by $\langle value, time, ImpAmt\rangle$. Here, *value* is an abstract data type that represents some characteristic value of the object type. The field *a.time* defines the age of attribute a. If an attribute a allows any amount of imprecision, then it must belong to a *metric space*. A metric space is a set of values on which a distance function is defined. The distance function has the properties of positivity and symmetry and it upholds the triangle inequality [12]. The field *a.ImpAmt* is the same type as *a.value*. It represents the amount of imprecision that has been introduced into the value of a. The attributes of the submarine include *Speed*, *Bearing* and *Country*. While *Speed* and *Bearing* may allow a certain amount of imprecision in their values (they are of the *real number* metric space), *Country* is not a metric space attribute and must therefore remain precise at all times.

An object type's M component is a set of methods that provides the only means for transactions to access instances of the object type. A method is defined by $\langle Arg, Op, Exec, OC\rangle$. *Arg* is a set of arguments each of which has the same structure as an attribute (*value*,*time*,*ImpAmt*). An *input* argument is one whose value is used by the method to update attributes. A *return* argument is one whose value is computed by the method and returned to the invoking transaction. We define the sets $InputArgs$ and $ReturnArgs$ to represent the subsets of Arg that contain the method's input arguments and the method's return arguments respectively. Op is a sequence of programming language operations, including reads and writes to attributes, that represents the executable code of the method.

$Exec$ is the worst case execution time of the method (run time on processor, not including blocking time). Although determining the worst case exeuction time of methods can, in general, be a difficult problem, techniques such as bounding loops, bounding recursion, and bounding dynamic memory allocation can make possible reasonable worst case bounds based on counting instruction executions [15]. An object-oriented method design technique [16] that encourages creating methods with bounded execution times will also facilitate this specification. User-specified and compiler-determined execution time bounding techniques are discussed in [15].

OC is a set of constraints on the execution of the method including absolute timing constraints on the method as a whole or on a subset of operations within the method [13]. In Figure 1 $IncPosition$ is a method of the **Submarine** object type which adds the value of its input argument to $Position.value$.

The C component of an object type is a set of constraints that defines correct states of an instance of the object type. A constraint is defined by $\langle Pr, ER\rangle$. Pr is a predicate which can include any of the three fields of attributes: value, time, and imprecision. Notice that both logical and temporal consistency constraints as well as bounds on imprecision can be expressed by these predicates. For instance, in Figure 1 the predicate $Speed.time > Now - 5$ expresses a temporal consistency constraint on the $Speed$ attribute requiring that it not be more than five seconds old. A logical constraint on $Speed$ is represented by the predicate $Speed.value >= 0$. The predicate $Speed.ImpAmt \leq 1.0$ defines the maximum amount of imprecision that may be allowed in the value of the $Speed$ attribute. The component ER of a constraint is an *enforcement rule* which is a sequence of programming language statements that is executed when the predicate becomes FALSE (*i.e.* when the constraint is violated).

The CF component of an object type is a boolean *compatibility function* with domain $M \times M \times SState$. The compatibility function uses semantic information about the methods as well as current system state ($SState$) to define compatibility between each ordered pair of methods of the object type. We describe the CF component in detail in Section 3.1.

2.2 Transactions

A transaction is defined by $\langle MI, L, C, P \rangle$. MI is a set of method invocation requests where each request is represented by $\langle M, Arg, temporal \rangle$. The M component of a method invocation request is an identifier for the method being invoked. Arg is the set of arguments to the method. Recall that a method argument can be a return argument or an input argument. A return argument $r \in Arg$ specifies a limit on the amount of imprecision allowed in the value returned through r as $import\_limit_r$. An input argument $i \in Arg$ specifies the value, time and imprecision amount to be passed to the method, as well as the maximum amount of imprecision that may be exported by the transaction through i, $export\_limit_i$. Note, the concurrency control technique we describe in Section 3 does not limit the amount of imprecision that a transaction may export. However, for generality, the model supports such a limit. The *temporal* field of a method invocation request specifies whether a transaction requires that temporally consistent data be returned.

The L component of a transaction is a set of lock requests and releases. Each lock request is associated with a method invocation request. A transaction may request a lock prior to the request for the method invocation, perhaps to enforce some transaction logical consistency requirement. In this case, the lock request is for a *future method invocation*. The transaction may also request the lock simultaneously with the method invocation, in which case the lock is requested for a *simultaneous method invocation*. This model of a transaction can achieve various forms of two-phase locking (2PL) [17] by requesting and releasing locks in specific orders. Other more flexible transaction locking techniques that do not follow 2PL can also be supported.

The component C of a transaction is a set of constraints on the transaction. These constraints can be expressed on execution, timing, or imprecision [13]. The priority P of a transaction is used by the database manager to perform real-time transaction scheduling (for a survey of real-time transaction scheduling see [9]). Each method invocation requested by the transaction is to be executed at the transaction's priority. Because a transaction is made up of a set of method invocations, our model assumes that a transaction cannot perform any intermediate computations.

Consider a situation in which a user of the submarine database wants precise location information on all submarines in the database. A transaction to perform such a task would request a lock and a simultaneous invocation of the *GetPosition* method on each submarine object in the database, specifying an imprecision import limit of zero for the arguments that return the locations. The transaction would hold the locks for these methods until all of the invocations are complete.

3 The Semantic Locking Technique

This section describes our real-time concurrency control technique for database objects under the RTSORAC model. The technique uses *semantic locks* to determine which transactions may invoke methods on an object. The granting of semantic locks is controlled by each individual object which uses its compatibility function to define conditional conflict. Our description of the technique concentrates on concurrency control within individual objects because we are concerned with bounding imprecision within objects and transactions. We briefly address inter-object concurrency control at the end of this section.

3.1 Compatibility Function

The compatibility function (CF) component of an object (Section 2.1) is a run-time function, defined on every ordered pair of methods of the object. The function has the form:

$$CF(m_{act}, m_{req}) =< BooleanExpression >$$

where m_{act} represents a method that has an active lock, and m_{req} represents a method for which a lock has been requested by a transaction.

The boolean expression may contain predicates involving several characteristics of the object or of the system in general. The concept of *affected set* that was introduced in [18], is used as a basis for representing the set of attributes of an object that a method reads/writes. We modify this notion to statically define for each method m a read affected set ($ReadAffected(m)$) and a write affected set ($WriteAffected(m)$). The compatibility function may refer to the *time* field of an attribute as well as the current time (Now) and the time at which an attribute a becomes temporally invalid ($deadline(a)$) to express a situation in which logical consistency may be traded-off to maintain or restore temporal consistency. The current amount of imprecision of an attribute a ($a.ImpAmt$) or a method's return argument r ($r.ImpAmt$) along with the limits on the amount of imprecision allowed on a ($data\_\epsilon_a$ [19]) and r ($import\_limit_r$) can be used to determine compatibility that ensures that interleavings do not introduce too much imprecision. The values of method arguments can be used to determine compatibility between a pair of method invocations, similar to techniques presented in [7].

Imprecision Accumulation. In addition to specifying compatibility between two locks for method invocations, the semantic locking technique requires that the compatibility function express information about the potential imprecision that could be introduced by interleaving method invocations. There are three potential sources of imprecision that the compatibility function must express for invocations of methods m_1 and m_2:

1. Imprecision in the value of an attribute that is in the write affected sets of both m_1 and m_2.
2. Imprecision in the value of the return arguments of m_1, when m_1 reads attributes written by m_2.
3. Imprecision in the value of the return arguments of m_2, when m_2 reads attributes written by m_1.

Compatibility Function Examples. Figure 2 uses the submarine example of Section 2.1 to demonstrate several ways in which the compatibility function can semantically express conditional compatibility of method locks. Example A shows how a compatibility function can express a trade-off of logical consistency for temporal consistency when a lock is currently active for *GetSpeed* and a lock on *UpdateSpeed* is requested. Under serializability, these locks would not be compatible because *GetSpeed*'s view of the *Speed* attribute could be corrupted. However, if the timing constraint on *Speed* is violated, it is important to allow *UpdateSpeed* to restore temporal consistency. Therefore, the two locks can be held concurrently as long as the value that is written to *Speed* by *UpdateSpeed* ($S_2.value$) is close enough to the current value of *Speed* (*Speed.value*). This determination is based on the imprecision limit of *GetSpeed*'s return argument S_1 and the amount of imprecision that *UpdateSpeed* will write to *Speed* through S_2 ($S_2.ImpAmt$). Also shown is the potential accumulation of imprecision that could result from the interleaving. In this case, *GetSpeed*'s return argument S_1 would have a potential increase in imprecision equal to the difference between the value of *Speed* before the update takes place (*Speed.value*) and the value of *Speed* after the write takes place ($S_2.value$), plus the amount of imprecision that is written to *Speed* by *UpdateSpeed* ($S_2.ImpAmt$).

Example B in Figure 2 illustrates how an attribute can become imprecise. Two invocations of *UpdateSpeed* may occur concurrently if a sensor writes one value and a human user, who has additional environmental information, also updates the *Speed*. Two locks on *UpdateSpeed* may be held concurrently as long as the difference between the values written by the associated invocations does not exceed the allowed amount of imprecision for the *Speed* attribute. In this case, the object's *Speed* attribute would have a potential increase in imprecision equal to the value of $|S_1.value - S_2.value|$ if this interleaving were allowed.

	Compatibility	Imprecision Accumulation
A:	$CF(GetSpeed(S_1), UpdateSpeed(S_2)) = (Speed.time < Now - 5)\ AND\ (\lvert Speed.value - S_2.value\rvert < import\_limit_{S_1} - (S_1.ImpAmt + S_2.ImpAmt))$	Increment $S_1.ImpAmt$ by $S_2.ImpAmt + \lvert Speed.value - S_2.value\rvert$
B:	$CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2)) = (\lvert S_1.value - S_2.value\rvert < data\_\epsilon_{Speed} - Speed.ImpAmt)$	Increment $Speed.ImpAmt$ by $\lvert S_1.value - S_2.value\rvert$
C:	$CF(IncPosition(A), GetPosition(P)) = \lvert A.value\rvert \leq import\_limit_P - P.ImpAmt$	Increment $P.ImpAmt$ by $\lvert A.value\rvert$

Figure 2: Compatibility Function Examples

Example C of Figure 2 represents a compatibility function involving a method that is more complex than the other examples. The method *IncPosition* reads the *Position* attribute, increments it by the value of input argument A and then writes the result back to the *Position* attribute. A lock for an invocation of this method may be held concurrently with a lock for an invocation of *GetPosition* only if the amount by which *IncPosition* increments the *Position* is within the imprecision bounds of the return argument P of *GetPosition*. In this case, *GetPosition*'s return argument P would have a potential increase in imprecision equal to the value of *IncPosition*'s argument A if this interleaving were allowed.

3.2 Semantic Locking Mechanism

The semantic locking mechanism must handle three actions by a transaction: a semantic lock request, a method invocation request and a semantic lock release. As described in Section 2.2, a semantic lock may be requested for a future method invocation request or for a simultaneous method invocation request. Future method invocation requests can be useful if a transaction requires that all locks be granted before any execution occurs, as with strict two-phase locking. Figures 3 and 4 show the procedures that the semantic locking mechanism executes when receiving a semantic lock request and a method invocation request respectively. A priority queue is maintained to hold requests that are not immediately granted.

3.2.1 Semantic Lock Request

When an object receives a semantic lock request for method invocation m_{req}, the semantic locking mechanism evaluates the compatibility function to ensure that m_{req} is compatible with all currently

```
Semantic Lock Request for $m_{req}$                             Step
granted := TRUE /* initialization */
for every (($m_{act} \in ActiveLocks$) OR                       LA
        (($m_{act}$ in priority queue) AND
        ($m_{act}.priority > m_{req}.priority$)))
    if $CF(m_{act}, m_{req})$ then                               $LA_1$
        save ImpAmts for return args of $m_{act}$
        Increment imprecision                                   $LA_2$
    else
        granted := FALSE
    endif
end for
if not granted then
    Enqueue($m_{req}$) in priority queue                        LB
else
    Add $m_{req}$ to $ActiveLocks$                              LC
endif
endif
```

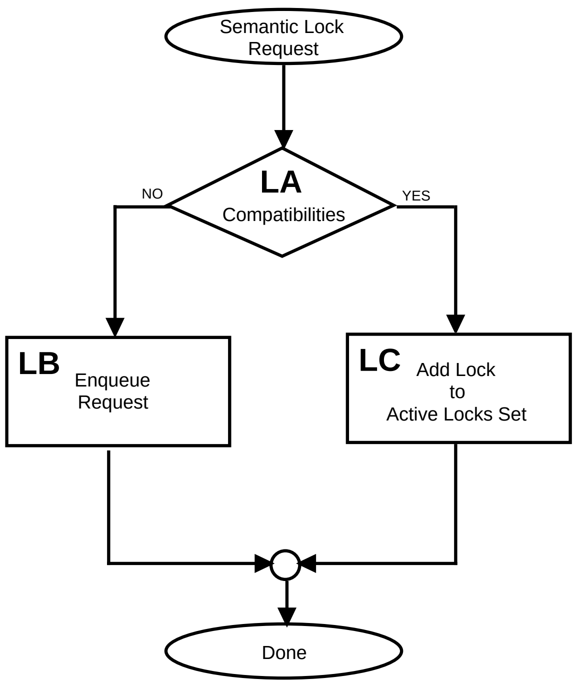

Figure 3: Mechanism for Semantic Lock Request

active locks and with all queued lock requests for method invocations that have higher priority than m_{req} (Figure 3, Step LA_1). For each compatibility function test that succeeds, the mechanism accumulates the imprecision that could be introduced by the corresponding interleaving (Step LA_2).

Recall that the boolean expression in the compatibility function can include tests involving value, time and imprecision information of the method arguments involved. A semantic lock request for a future method invocation does not have values for arguments at the time of the request. Thus, when evaluating the compatibility function for $CF(m_{act}, m_{req})$, if either m_{act} or m_{req} is a future method invocation, then any clause of the compatibility function that involves method arguments must evaluate to FALSE.

If all compatibility function tests succeed, the semantic locking mechanism grants the semantic lock and places it in the active lock set (Step LC). If any test fails, the mechanism places the request in the priority queue to be retried when another lock is released (Step LB).

3.2.2 Method Invocation Request

When an object receives a method invocation request, the semantic locking mechanism evaluates a set of preconditions and either requests a semantic lock for the invocation if necessary or updates the existing semantic lock with specific argument amounts. After the preconditions are successfully

```
Method Invocation Request for $m_{req}$                         Step
InitialImprecision($m_{req}$)                                   A
if any Precondition fails then                                  B
    Enqueue($m_{req}$) in priority queue                        L
else
for every $a \in WriteAffected(m_{req})$                        C$_1$
    save original $a.ImpAmt$
    $a.ImpAmt := m_{req}.ExportImp(a)$
end for
for every $r \in ReturnArgs(m_{req})$                           C$_2$
    save original $r.ImpAmt$
    $r.ImpAmt := m_{req}.ImportImp(r)$
end for
if already locked then                                          D
    Allow Execution of $m_{req}$                                I
    Semantic Lock Update                                        J
    Check the queue                                             K
else
    Semantic Lock Request                                       E
    if lock granted then                                        F
        Allow Execution of $m_{req}$                            H
    else
        for every $a \in WriteAffected(m_{req})$                G
            restore original $a.ImpAmt$
        for every $r \in ReturnArgs(m_{req})$
            restore original $r.ImpAmt$
        for every saved return argument $r$
          of an active method invocation
            restore original $r.ImpAmt$
endif
endif
```

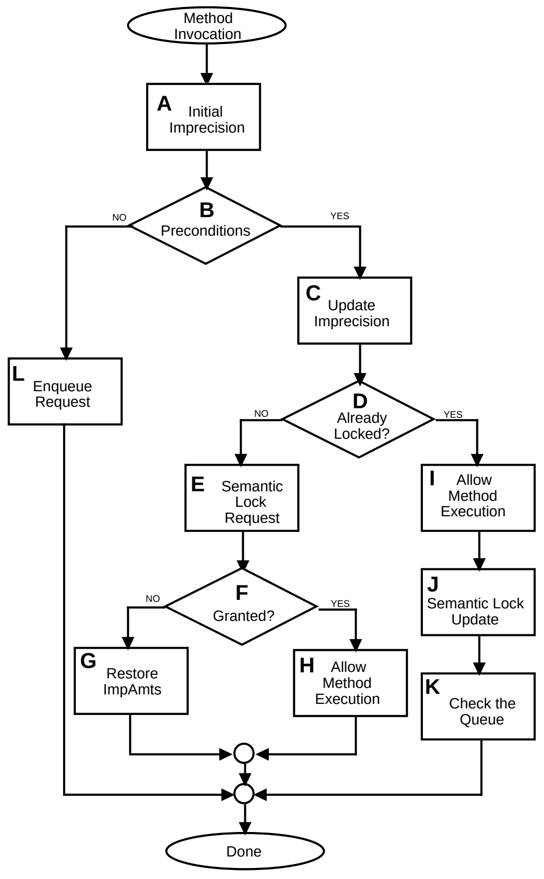

Figure 4: Mechanism for Method Invocation Request

evaluated and locks are granted or updated, the semantic locking mechanism allows the method invocation to execute. The mechanism also accumulates the imprecision that could result if the requested method were to execute. In the following paragraphs we describe the steps in Figure 4 of the semantic locking mechanism for a method invocation request m_{req}.

Initial Imprecision Calculation. Given method invocation request m_{req}, the semantic locking mechanism first computes the potential amount of imprecision that m_{req} will introduce into the attributes that it writes and into its return arguments. This computation takes into account the imprecision in the attributes read by the methods and in the input arguments as well as any computations that are done by the method on these values (Figure 4, Step A). An initial imprecision procedure computes the amount of imprecision that m_{req} will write to each attribute a in the write affected set of m_{req} ($m_{req}.ExportImp(a)$). The procedure also computes the amount of imprecision that m_{req} will return through each of its return arguments r ($m_{req}.ImportImp(r)$). The procedure computes these values by using the amount of imprecision already in the attribute or return argument and calculating how the method may update this imprecision through operations that it performs. This initial imprecision procedure may be created by the object designer or by a compile-time tool that examines the structure of m_{req} to determine how the method will affect the imprecision of attributes in its write affected set and of its return arguments.

Preconditions Test. The next phase of the semantic locking mechanism for method invocation request m_{req} tests preconditions that determine if executing m_{req} would violate temporal consistency or imprecision constraints (Step B). The mechanism evaluates the following preconditions when m_{req} has been requested:

Preconditions

$$m_{req}.temporal \Longrightarrow (\forall_{a \in ReadAffected(m_{req})}(Exec(m_{req}) < deadline(a) - Now)) \qquad (a)$$

$$\forall_{a \in WriteAffected(m_{req})}(m_{req}.ExportImp(a) \leq data\_\epsilon_a) \qquad (b)$$

$$\forall_{r \in ReturnArgs(m_{req})}(m_{req}.ImportImp(r) \leq import\_limit_r) \qquad (c)$$

Precondition (a) ensures that if a transaction requires temporally valid data, then an invoked method will not execute if any of the data that it reads will become temporally invalid during its execution time. Precondition (b) ensures that executing the method invocation will not allow too much initial imprecision to be introduced into attributes that the method invocation writes.

Precondition (c) ensures that the method invocation executes only if it does not introduce too much initial imprecision into its return arguments.

If any precondition fails, then the semantic locking mechanism places the request on the priority queue (Step L) to be retried when another lock is released. If the preconditions hold, the semantic locking mechanism updates the imprecision amounts for every attribute a in the write affected set of m_{req} with the value $m_{req}.ExportImp(a)$. Similarly, it updates the imprecision amounts for every return argument r of m_{req} with the value $m_{req}.ImportImp(r)$ (Step C). The mechanism saves the original values for the imprecision amounts of the attributes and return arguments involved so that they can be restored if the lock is not granted.

Because the preconditions can block a transaction if the data that it accesses is too imprecise for its requirements, there must be some way of restoring precision to data so that transactions are not blocked indefinitely. Certain transactions that write precise data are characterized as *independent updates* [19]. Such a transaction, which may come from a sensor or from user intervention, restores precision to the data that it writes and allows transactions that are blocked by the imprecision of the data to be executed.

Associated Semantic Lock. The semantic locking mechanism next determines whether or not m_{req} is already locked by a semantic lock requested earlier (Step D). If not, a semantic lock is requested (Step E) as described in Section 3.2.1. If the lock is granted, the semantic locking mechanism allows the method invocation to execute (Step H). Otherwise, the mechanism restores the original values of any imprecision amounts that were changed (Step G).

If the semantic lock associated with m_{req} was granted earlier, the semantic locking mechanism allows m_{req} to be executed (Step I). The mechanism then performs a semantic lock update (Step J). This procedure entails updating the existing semantic lock associated with m_{req} with specific argument information that was not available when the lock was granted. Updating existing locks potentially increases concurrency among methods because with values of arguments, the compatibility function is more likely to evaluate to TRUE. After the semantic lock is updated, the lock requests waiting on the priority queue are checked for compatibility with the newly updated lock (Step K).

3.2.3 Releasing Locks.

A semantic lock is released explicitly by the holding transaction. Whenever a semantic lock is released, it is removed from the active locks set and the priority queue is checked for any requests that may be granted. Since the newly-released semantic lock may have been associated with a method invocation that restored logical or temporal consistency to an attribute, or the lock may have caused some incompatibilities, some of the queued lock requests may now be granted. Also, method invocation requests that are queued may now pass preconditions if temporal consistency or precision has been restored to the data. The requests in the queue are re-issued in priority order and if any of these requests is granted, it is removed from the queue.

3.3 Inter-Object Concurrency Control

The semantic locking mechanism described in this paper maintains consistency for individual objects and transactions. In addition, transactions in the current technique can obtain multiple locks and therefore can enforce inter-object consistency themselves. This enforcement is similar to techniques used in traditional database systems – it requires that transactions are written to maintain inter-object consistency.

Extending semantic locking to provide system enforcement of inter-object consistency is possible, but outside the scope of this paper. We outline the approach here. As mentioned in Section 2, inter-object constraints are expressed in RTSORAC *relationships*. An inter-object constraint is defined on the methods of the objects participating in the relationship and is enforced by the *enforcement rule* of the constraint. An enforcement rule of an inter-object constraint may invoke methods of the participating objects. Thus, to automatically support an inter-object constraint, the semantic locking technique should propagate semantic lock requests through relationships to ensure that the enforement rule that maintains the inter-object constraint can execute. For instance, assume that a semantic lock is requested on a method m_1 of an object o_1 that participates in relationship r. Relationship r has an inter-object constraint c between o_1 and an object o_2. The enforcement rule of constraint c requires that a method m_2 be executed under some conditions of m_1's execution. So, upon a request for a semantic lock on m_1, the semantic locking mechanism should also propagate a semantic lock request for m_2 to o_2. All propagated locks should be granted before the original lock request is granted. Propagated semantic locks would be released when the original lock is released.

This paper concentrates on semantic locking and imprecision management for individual objects, which is a significant problem. We are working on extending the semantic locking technique to automatically support inter-object constraints as outlined here, but a further description is outside the scope of this paper.

3.4 Implementation

We have implemented the RTSORAC model in a prototype system that extends the Open Object Oriented Database System (Open OODB) [20] to support real-time requirements. These real-time extensions execute on a Sun Sparc Classic workstation under the Solaris 2.4 operating system. RTSORAC objects are implemented in main memory using Solaris' shared memory capability. Transactions can access objects in the shared memory segment as if the objects were in their own address space. This design provides fast, predictable access to data objects. Before accessing objects, transactions execute the semantic locking mechanism to provide concurrency control. Performance measurements on the prototype system indicate that requesting a semantic lock requires approximately 60 μs if there are no other locks on the object. This time increases linearly for each active lock and each pending request. The implementation is described fully in [21].

We have completed some preliminary performance tests using the prototype implementation. The tests compared our semantic locking technique with other lock-based concurrency control techniques, such as object locking, read/write locking and commutativity-based locking. Overall, the results indicated that our semantic locking technique allowed transactions to meet as many if not more deadlines than the other techniques tested. For a full description of the performance testing and results, see [22].

4 Bounding Imprecision

In this section we show how our semantic locking technique can bound imprecision in the objects and transactions of the database. To do this, we prove that the semantic locking technique, under two general restrictions on the design of each object's compatibility function, ensures that the epsilon-serializability (ESR) [12] correctness criteria, defined for object-oriented databases, is met. First, we summarize the definition of ESR from [12, 19] and then extend its definition to object-oriented databases. Second, we present the two general restrictions on the compatibility function. Third, we formally prove the sufficiency of these restrictions for ensuring that our semantic locking technique maintains object-oriented ESR. Finally, we describe an example of how the restricted semantic locking technique bounds imprecision in the submarine tracking example.

4.1 Epsilon Serializability

Epsilon serializability (ESR) is a correctness criterion that generalizes serializability by allowing bounded imprecision in transaction processing. ESR assumes that serializable schedules of transactions using precise data always result in precise data in the database and in precise return values from transactions. A value resulting from a schedule H is *imprecise* if it differs from the corresponding value resulting from each possible serializable schedule of the transactions in H. In order to accumulate and limit imprecision, ESR assumes use of only data items that belong to a *metric space* (defined in Section 2) [12].

A transaction specifies limits on the amount of imprecision that it can import and export with respect to a particular data item. $Import\_limit_{t,x}$ is defined as the maximum amount of imprecision that transaction t can import with respect to data item x, and $export\_limit_{t,x}$ is defined as the limit on the amount of imprecision exported by transaction t to data item x [12]. For every data item x in the database, a data ϵ-specification ($data\_\epsilon_x$) expresses a limit on the amount of imprecision that can be written to x [19].

The amount of imprecision imported and exported by each transaction, as well as the imprecision written to the data items, must be accumulated during the transaction's execution. $Import\_imprecision_{t,x}$ represents the amount of imprecision imported by transaction t with respect to data item x. Similarly, $export\_imprecision_{t,x}$ represents the amount of imprecision exported by transaction t with respect to data item x. $Data\_imprecision_x$ defines the amount of imprecision written to the data item x.

ESR defines $Safety$ as a set of conditions that specifies boundaries for the amount of imprecision permitted in transactions and data. Safety is divided into two parts: transaction safety and data safety. Safety for transaction t with respect to data item x is defined in [12] as follows:[1]

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

Data safety is described informally in [19]. We formalize the definition of data safety for data item x:

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

The original definition of ESR [12, 19] can now be stated as: ESR is guaranteed if and only if all transactions and data items are safe. Or, more formally as:

Definition 1 *ESR is* guaranteed *if and only if* $TR\text{-}Safety_{t,x}$ *and* $Data\text{-}Safety_x$ *are invariant for every transaction* t *and every data item* x.

[1]In [12] the terms $import\_inconsistency_{t,x}$ and $export\_inconsistency_{t,x}$ are used. We have renamed them to $import\_imprecision_{t,x}$ and $export\_imprecision_{t,x}$.

It is this definition that we adapt for the object-oriented data model and use to show that our semantic locking technique maintains bounded imprecision.

4.2 Object-Oriented ESR

The above definitions of data and transaction safety were general; we now define safety more specifically for the RTSORAC real-time object-oriented database model. Although this model allows arbitrary attributes and return arguments, we assume in the following definitions and theorem that each attribute value is an element of some metric space.

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

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

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

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

$$\mathit{OT\text{-}Safety}_{t,o} \equiv \left\{ \begin{array}{l} \forall_{m \in (t_{MI} \sqcap o_M)} \forall_{r \in ReturnArgs(m)} (r.\mathit{ImpAmt} \leq \mathit{import\_limit}_r) \\ \forall_{m \in (t_{MI} \sqcap o_M)} \forall_{i \in InputArgs(m)} (i.\mathit{ImpAmt} \leq \mathit{export\_limit}_i) \end{array} \right.$$

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

We can now define Object Epsilon Serializability (OESR) as:

Definition 2 *OESR is* guaranteed *if and only if* $\mathit{OT\text{-}Safety}_{t,o}$ *and* $\mathit{Object\text{-}Safety}_o$ *are invariant for every object transaction* t *and every object* o.

This definition of OESR is a specialization of the general definition of ESR.

4.3 Restrictions on The Compatibility Function

The RTSORAC compatibility function allows the object type designer to define compatibility among object methods based on the semantics of the application. We now present two restrictions on the conditions of the compatibility function that are sufficient to guarantee OESR. Intuitively, these restrictions allow read/write and write/write conflicts over affected sets of methods as long as specified imprecision limits are not violated.

The imprecision that is managed by these restrictions comes from interleavings allowed by the compatibility function. Any imprecision that may be introduced by calculations performed by the methods is accumulated the initial imprecision procedure before the compatibility function is evaluated (see Section 3.2.2).

Let a be an attribute of an object o, and m_1 and m_2 be two methods of o.

Restrictions

R1: If $a \in WriteAffected(m_1) \bigcap WriteAffected(m_2)$ then the compatibility function for $CF(m_1, m_2)$ and $CF(m_2, m_1)$ may return TRUE only if it includes the conjunctive clause: $|z_1 - z_2| \leq (data\_\epsilon_a - a.ImpAmt)$, where z_1 and z_2 are the values written to a by m_1 and m_2 respectively. Furthermore, the compatibility function's associated imprecision accumulation must specify the following for a: $a.ImpAmt := a.ImpAmt + |z_1 - z_2|$.

R2: If $a \in ReadAffected(m_1) \bigcap WriteAffected(m_2)$ then for every $r \in ReturnArgs(m_1)$ let z be the value of r using a's current value, let x be the value written to a my m_2 and let w be the value of r using x. Then:

- **a)** the compatibility function for $CF(m_2, m_1)$ may return TRUE only if it includes the conjunctive clause: $|z - w| \leq (import\_limit_r - r.ImpAmt)$. Furthermore, the compatibility function's associated imprecision accumulation must specify the following for r: $r.ImpAmt := r.ImpAmt + |z - w|$.
- **b)** the compatibility function for $CF(m_1, m_2)$ may return TRUE only if it includes the conjunctive clause: $|z - w| \leq (import\_limit_r - (r.ImpAmt + x.ImpAmt))$. Furthermore, the compatibility function's associated imprecision accumulation must specify the following for r: $r.ImpAmt := r.ImpAmt + x.ImpAmt + |z - w|$.

Restriction R1 captures the notion that if two method invocations interleave and write to the same attribute a, the amount of imprecision that may be introduced into a is at most the distance between the two values that are written ($|z_1 - z_2|$). To maintain safety, this amount cannot be greater than the imprecision limit less the current amount of imprecision for a ($data\_\epsilon_a - a.ImpAmt$). The accumulation of this imprecision in $a.ImpAmt$ is also reflected in R1.

As an example of restriction R1, recall the compatibility function example of Figure 2B of Section 3.1. Notice that the *Speed* attribute is in the write affected set of the method *UpdateSpeed* and thus restriction R1 applies to the compatibility function $CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2))$. The value written to the *Speed* attribute by $UpdateSpeed_1$ is S_1 and the value written to *Speed* by $UpdateSpeed_2$ is S_2. Thus, the compatibility function, $CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2))$ may return TRUE only if $|S_1 - S_2| \leq (data\_\epsilon_{Speed} - Speed.ImpAmt)$.

Restriction R2 is based on the fact that if a method invocation that reads an attribute (m_1) is interleaved with a method invocation that writes to the same attribute (m_2), the view that m_1 has of the attribute (in return argument r) may be imprecise. In R2a the amount of imprecision in m_1's view of the attribute is at most the distance between the value of the attribute before m_2's write takes place and the value of the attribute after m_2's write takes place ($|z - w|$). This amount cannot be greater than the imprecision limits imposed on r less the current amount of imprecision on r ($import\_limit_r - r.ImpAmt$); it also must be accumulated in the imprecision amount of r.

Restriction R2b differs from R2a in that when R2b applies, m_1 is currently active and m_2 has been requested. The initial imprecision procedure for m_1 computes the amount of imprecision that m_1 will return through r ($m_1.ImportImp(r)$) before m_2 is invoked, and thus $r.ImpAmt$ does not include the amount of imprecision that m_2 might introduce into a ($x.ImpAmt$). Because allowing the interleaving between m_1 and m_2 could cause any imprecision introduced into a to be returned by m_1 through r, the additional amount of imprecision introduced to a by m_2 ($x.ImpAmt$) must be taken into account when testing for compatibility between m_1 and m_2. It must also be included in the accumulation of imprecision for r.

Figure 2A of Section 3.1 presents an example of a compatibility function that meets restriction R2b. Notice that the function will evaluate to TRUE only if the difference between the value of the *Speed* attribute before the update takes place ($Speed.value$) and the value of the attribute after the update takes place ($S_2.value$) is within the allowable amount of imprecision specified for the return argument of the *GetSpeed* method. Notice also that this allowable amount of imprecision must take into account the amount of imprecision already in the return argument ($S_1.ImpAmt$) and the amount of imprecision in the argument used to update the *Speed* attribute ($S_2.ImpAmt$).

Each of the restrictions requires that non-serializable interleavings be allowed only if certain conditions involving argument amounts evaluate to TRUE. Thus, for $CF(m_1, m_2)$, if either m_1 or m_2 is a future method invocation, then the restrictions require that only serializable interleavings be allowed because argument values of future method invocations are not known. Therefore, no imprecision will be accumulated when one or both method invocations being tested for compatibility is a future method invocation.

We call the concurrency control technique that results from placing Restrictions R1 and R2 on the compatibility function, *the restricted semantic locking technique.*

4.4 Correctness

We now show how the restricted semantic locking technique guarantees OESR. First, we prove a lemma that Object-Safety remains invariant through each step of the semantic locking mechanism. We then prove a similar lemma for OT-Safety. Both of these lemmas rely on the design of the restricted semantic locking technique, which contains tests for safety conditions before each potential accumulation of imprecision.

It is sufficient to demonstrate that safety is maintained for semantic lock requests for simultaneous method invocations only, since this is the only part of the semantic locking mechanism that can introduce imprecision into data and transactions. A semantic lock request for a future method invocation m does not introduce imprecision because the argument amounts are not known. Thus restrictions R1 and R2 require that no imprecision be accumulated when interleaving m with any other method invocation. Lock releases also do not introduce imprecision.

Lemma 1 *If the restricted semantic locking technique is used, then $Object\text{-}Safety_o$ is invariant for every object o.*
Proof:

> Let o be an object and o_A be the set of attributes in o. We assume that o is initially safe and that the restricted semantic locking technique is used. Consider the steps in the semantic locking mechanism (Figure 4) in which the imprecision amount of a, $a.ImpAmt$, is updated:
>
> - (Step C) Imprecision is accumulated if the preconditions for a requested method invocation m hold and $a \in WriteAffected(m)$. Since the preconditions hold, Step C_1 ensures $a.ImpAmt = m.ExportImp(a)$, and from Precondition (b): $m.ExportImp(a) \leq data\_\epsilon_a$. Combining these two relations we have that $a.ImpAmt \leq$

$data\_\epsilon_a$, which is the requirement for Object Safety. Thus, Object Safety remains invariant after Step C.

- (Step LA) Imprecision is accumulated in Step LA$_2$ if the compatibility function evaluation in Step LA$_1$ for method invocations m_1 and m_2 evaluates to TRUE and $a \in WriteAffected(m_1) \bigcap WriteAffected(m_2)$. In this case, the imprecision after Step LA$_2$ is $a.ImpAmt_{new} = a.ImpAmt_{old} + |z_1 - z_2|$, where z_1 and z_2 are the values written to a by m_1 and m_2 respectively. From Restriction R1 we have that $|z_1 - z_2| \leq data\_\epsilon_a - a.ImpAmt_{old}$. This inequality can be rewritten as $a.ImpAmt_{old} + |z_1 - z_2| \leq data\_\epsilon_a$. Combining this relation with the above relation involving $a.ImpAmt_{new}$ yields: $a.ImpAmt_{new} \leq data\_\epsilon_a$, which is the requirement for Object Safety. Thus, Object Safety remains invariant after Step LA. $\square$

Lemma 2 *If the restricted semantic locking technique is used, then $OT\text{-}Safety_{t,o}$ is invariant for every transaction t with respect to every object o.*
Proof:

Let o be an object, t be a transaction, m be a method invocation on o invoked by t, r be a return argument of m, and i be an input argument of m. We assume that t is initially safe with respect to o and that the restricted semantic locking technique is used. We show that $r.ImpAmt \leq import\_limit_r$ first for the case when a semantic lock for m is requested by t and then for the case when t holds the semantic lock for m.

Case 1. Transaction t requests a semantic lock for m and a semantic lock is held for another method invocation m_1. Consider the situations in which $r.ImpAmt$ is updated by the semantic locking mechanism:

- (Step C) Imprecision is accumulated if the preconditions for m hold. Since the preconditions hold, Step C$_2$ ensures $r.ImpAmt = m.ImportImp(r)$, and from Precondition (c): $m.ImportImp(r) \leq import\_limit_r$. Combining these two relations we have that $r.ImpAmt \leq import\_limit_r$, which is the requirement for OT Safety. Thus, OT Safety remains invariant after Step C.
- (Step LA) Imprecision is accumulated in Step LA$_2$ if the compatibility function evaluation in Step LA$_1$ for $CF(m_1, m)$ evaluates to TRUE and $ReadAffected(m) \bigcap WriteAffected(m_1) \neq \emptyset$. In this case, the imprecision after Step LA$_2$ is $r.ImpAmt_{new} = r.ImpAmt_{old} + |z - w|$, where z is the value of r using the

current value of a, and w is the value of r using the value written by m_1 to a. From Restriction R2a we have that $|z - w| \leq import\_limit_r - r.ImpAmt_{old}$. This inequality can be rewritten as $r.ImpAmt_{old} + |z - w| \leq import\_limit_r$. Combining this relation with the above relation involving $r.ImpAmt_{new}$ yields: $r.ImpAmt_{new} \leq import\_limit_r$, which is the requirement for OT Safety. Thus, OT Safety remains invariant after Step LA. □

Case 2 Transaction t holds the semantic lock for m and a semantic lock is requested for m_1. In this case, $r.ImpAmt$ can only be updated in Step LA of the semantic locking mechanism and only when the compatibility function evaluation in Step LA$_1$ for $CF(m, m_1)$ evaluates to TRUE and $ReadAffected(m) \bigcap WriteAffected(m_1) \neq \emptyset$. In this case, the imprecision after Step LA$_2$ is $r.ImpAmt_{new} = r.ImpAmt_{old} + x.ImpAmt + |z - w|$, where x is value written to a by m_1, z is the value of r using a's current value and w is the value of r using x. From Restriction R2b we have that $|z - w| \leq import\_limit_r - (r.ImpAmt_{old} + x.ImpAmt)$. This inequality can be rewritten as $r.ImpAmt_{old} + x.ImpAmt + |z - w| \leq import\_limit_r$. Combining this relation with the above relation involving $r.ImpAmt_{new}$ yields: $r.ImpAmt_{new} \leq import\_limit_r$, which is the requirement for OT Safety. Thus, OT Safety remains invariant after Step LA. □

The other OT safety property, $i.ImptAmt \leq export\_limit_i$, is trivially met because the semantic locking technique does not limit the amount of imprecision that is exported by a transaction to other transactions or to objects. As stated in [19], if transactions execute simple operations, the export limit can be omitted and the transaction can rely completely on $data\_\epsilon$ specifications for imprecision control. The simple model of transactions of Section 2.2 allows us to define for all input arguments i, $export\_limit_i = \infty$. Thus, regardless of the value of $i.ImpAmt$, OT safety is invariant. □

Theorem 1 *If the restricted semantic locking technique is used, then OESR is guaranteed.*
Proof: Follows from Definition 2, Lemma 1, and Lemma 2. □

Theorem 1 shows that if the restricted semantic locking technique is used, the imprecision that is introduced into the data and transactions is bounded. Because OESR is guaranteed across all objects and all transactions, this result shows that the restricted semantic locking technique maintains a single, global correctness criterion that bounds imprecision in the database.

4.5 Example

We use an example of a **Submarine** object, which is an instance of the object type in Figure 1 of Section 2, to illustrate how the semantic locking technique maintains the imprecision limits of a data object and therefore guarantees OESR. The object's method $UpdateSpeed(S)$ writes the value S to the value field of the object's $Speed$ attribute. We assume that the $Speed$ attribute is initially precise ($Speed.ImpAmt = 0$), that the only active lock is for a simultaneous invocation of $UpdateSpeed(10.0)$, and that the object's priority queue is empty. Let a transaction request a lock for a simultaneous invocation of $UpdateSpeed(10.6)$, where the value 10.6 has 0.3 units of imprecision in it. As indicated in Figure 1, the imprecision limit on $Speed$ is $data\_\epsilon_{Speed} = 1.0$.

When the **Submarine** object receives the request for the $UpdateSpeed(10.6)$ method invocation it executes the semantic locking mechanism of Figure 4. First it computes the initial imprecision procedure (Step A). $Speed$ is the only attribute in the write affected set of $UpdateSpeed$ and $UpdateSpeed$ has no return arguments, so the initial imprecision procedure computes $UpdateSpeed.ExportImp(Speed)$. Because the invocation $UpdateSpeed(10.6)$ writes 10.6 to $Speed$ with 0.3 units of imprecision, $UpdateSpeed.ExportImp(Speed) = 0.3$.

The preconditions for the requested $UpdateSpeed(10.6)$ method invocation are tested next (Step B). Precondition (a) trivially holds because $ReadAffected(UpdateSpeed) = \emptyset$. Precondition ($b$) also holds since $UpdateSpeed.ExportImp(Speed) = 0.3 \leq 1.0$. Since $UpdateSpeed$ has no return arguments, Precondition (c) holds as well.

Step C_1 of the semantic locking mechanism then initializes the imprecision amount for the $Speed$ attribute to the value of $UpdateSpeed.ExportImp(Speed)$, so $Speed.ImpAmt = 0.3$.

Because the semantic lock was requested for a simultaneous method invocation, the condition in Step D is TRUE and a semantic lock request is performed (Step E). In Step LA_1, the object's semantic locking mechanism checks the compatibility of the requested invocation of $UpdateSpeed(10.6)$ with the currently locked invocation of $UpdateSpeed(10)$. Recall from Figure 2 and the example in Section 4.3 that $CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2)) = |S_1.value - S_2.value| \leq data\_\epsilon_{Speed} - Speed.ImpAmt$. The test of the compatibility function uses the imprecision amount for $Speed$ that was stored in Step C and thus: $|S_1.value - S_2.value| = |10 - 10.6| = 0.6$ and $data\_\epsilon_{Speed} - Speed.ImpAmt = 1.0 - 0.3 = 0.7$. Since $0.6 \leq 0.7$, the method invocations are compatible in Step LA_1.

Now the object's semantic locking mechanism executes Step LA_2 to accumulate imprecision for the $Speed$ attribute into the imprecision amount for $Speed$ stored in Step C. Recall from Figure

2: $CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2)) \Rightarrow Speed.ImpAmt := Speed.ImpAmt + |S_1.value - S_2.value|$. Thus, the mechanism computes a new value for the imprecision amount for the *Speed* attribute as: $Speed.ImpAmt := 0.3 + 0.6 = 0.9$.

Because there are no other active locks to check for compatibility, the compatibility function evaluates to TRUE. The object's mechanism grants a semantic lock for the invocation of *UpdateSpeed*(10.6) and adds the lock to the object's active lock set (Step LC). Finally the semantic locking mechanism allows the execution of *UpdateSpeed*(10.6) (Step H). Note that the imprecision amount for the *Speed* attribute is now 0.9. Both *UpdateSpeed* method invocations execute concurrently and the imprecision limits are maintained.

Although we have only demonstrated relatively simple method interleavings in this example (essentially two writes to a single attribute), the use of read affected and write affected sets in the semantic locking technique allows it to perform in a similar fashion for more complicated object methods.

5 Conclusion

This paper has presented a model and an object-based semantic real-time concurrency control technique capable of enforcing both temporal and logical consistency constraints within real-time database objects. Moreover, it demonstrated that the technique can bound the imprecision that is introduced when one constraint is traded off for another. This was done by showing that, under certain general restrictions, the technique guarantees a global correctness criterion – a specialization of epsilon serializability for object-oriented databases.

Although our technique is designed for soft real-time databases and therefore offers no guarantees of meeting timing constraints, the support that it provides for real-time is in its treatment of temporal consistency requirements. The user-defined compatibility function provides support for maintaining data temporal consistency by allowing the specification of the trade-off between temporal and logical consistency. Because our technique allows for relaxing serializability among transactions, the likelihood that the real-time scheduler will be able to determine a schedule that maintains transaction timing constraints is increased.

Our technique differs from most previous real-time concurrency control work [9] and the semantic concurrency control work in [2, 5, 6] because it is based on an object-oriented data model. It differs from the object-based concurrency control work in [7, 8, 11, 18, 23] because it incorporates temporal consistency requirements. It differs from all of these approaches and the real-time concurrency control work in [10] in that it can manage and bound imprecision that can be accumulated due to trading off logical consistency for temporal consistency. It differs from other ESR-based techniques [24] because it can limit logical imprecision to be introduced only if temporal consistency of data or transactions is threatened.

Our semantic locking technique is closest to the concurrency control protocol presented in [8]. This protocol uses commutativity with bounded imprecision to define operation conflicts. It is similar to our protocol in that the user defines the allowed amount of imprecision for a given operation invocation and the protocol uses a modified commutativity table to determine if the operation can execute concurrently with the active operations. However, the protocol in [8] does not take temporal considerations into account. Furthermore, our restrictions on the compatibility function, defined in Section 4.3, provide the user with a guide for defining the compatibility function to maintain correctness. There is no similar guide in [8] for defining commutativity with bounded imprecision.

Two drawbacks of our technique are the complexity posed to the system designer and the additional overhead required for the run-time system to grant locks. One reason for the complexity is that applications that require real-time database management, such as submarine command and control, are generally more complex than those that can be supported by traditional databases. Adding support for imprecision maintenance, while providing a potential increase in concurrency, also adds to the complexity of the technique. We are currently developing a tool to ease some of the burden on system designers. The tool computes read and write affected sets of methods, along with other static characteristics, and proposes default object compatibility functions and imprecision accumulation. The designer can then interactively modify the compatibility function or the constraints of objects or transactions based on application-specific semantic information.

Although the performance measurements for our technique in our prototype system indicate that it takes on the order of hundreds of microseconds (depending on the number of current locks and requests) to execute semantic locking, the extra overhead is not prohibitive. It does indicate that semantic locking is not appropriate for applications with short method executions and lock durations. For longer-lived method executions and transactions, the increased concurrency of semantic locking will easily justify the increased overhead. Unfortunately, bounding the overhead and the blocking time that are introduced by the semantic locking technique is not feasible due to the complexity of the technique; this limits its usefulness in hard real-time databases. However, the results of preliminary performance studies comparing the semantic locking with other lock-based concurrency control techniques, indicate that the semantic locking technique is generally better at meeting timing constraints than the other techniques tested.

We believe that the generality of our technique (a conditional compatibility function and semantic locking mechanism distributed to each object), the treatment of temporal consistency, the definition of restrictions that are sufficient to bound imprecision, and the definition of an object-oriented version of ESR, are valuable contributions towards expressing and enforcing imprecision in object databases as well as providing support for maintaining both temporal and logical consistency found in real-time databases.

Acknowledgements. We thank Joan Peckham, Janet Prichard, Paul Fortier, and Krithi Ramamritham for their helpful comments and suggestions. We thank John Black for his work in implementing the prototype system and his feedback along the way.

References

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

[2] 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.

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

[4] N. S. Barghouti and G. E. Kaiser, "Concurrency control in advanced database applications," *ACM Computing Surveys*, vol. 23, pp. 269–316, September 1991.

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

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

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

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

[9] P. S. Yu, K.-L. Wu, K.-J. Lin, and S. H. Son, "On real-time databases: Concurrency control and scheduling," *Proceedings of the IEEE*, vol. 82, pp. 140–157, January 1994.

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

[11] R. F. Resende, D. Agrawal, and A. E. Abbadi, "Semantic locking in object-oriented database systems," in *9th OOPSLA*, October 1994.

[12] K. Ramamritham and C. Pu, "A formal characterization of epsilon serializability," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, October 1995.

[13] J. Prichard, L. C. DiPippo, J. Peckham, and V. F. Wolfe, "*RTSORAC*: A real-time object-oriented database model," in *The 5th International Conference on Database and Expert Systems Applications*, Sept. 1994.

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

[15] W. Pugh and T. M. (Editors), "Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems," June 1994. held in conjunction with ACM SIGPLAN PLDI Conference.

[16] J. Rumbaugh, "Relations as semantic constructs in object-oriented languages," in *ACM OOPSLA Proceedings*, pp. 466–481, October 1987.

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

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

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

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

[21] V. F. Wolfe, L. C. DiPippo, J. Prichard, and J. Peckham, "The design of real-time extensions to the open object-oriented database system," in *The 1st IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Oct. 1994.

[22] L. C. DiPippo and V. F. Wolfe, "Testing the performance of object-based semantic real-time concurrency control," Tech. Rep. TR95-245, Department of Computer Science and Statistics, University of Rhode Island, 1995.

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

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