Towards Priority Ceilings in Object-Based Semantic Real-Time Concurrency Control*

Michael Squadrito and Bhavani Thuraisingham
MITRE Corporation
Bedford, MA USA
thura@mitre.org

Lisa Cingiser DiPippo and Victor Fay Wolfe
Department of Computer Science
University of Rhode Island USA
{squadrit,dipippo,wolfe}@cs.uri.edu

Abstract

This paper shows how priority ceiling techniques can be added to object-based real-time semantic concurrency control. The resulting protocol provides more potential concurrency for real-time object-oriented databases than previous priority ceiling techniques, while alleviating priority inversion and deadlock problems of previous object-based semantic concurrency control techniques. It is also a natural extension of priority ceiling techniques to objects in general.

1 Introduction

In real-time databases *real-time concurrency control* must synthesize two sets of traditional requirements: those for real-time and those for concurrency control. Among the requirements imposed by real-time applications are those for fast execution and/or predictable execution. Among the requirements imposed by concurrency control is the need to enforce logical consistency constraints in the database.

A popular form of real-time concurrency control has been to augment lock-based concurrency control with *priority inheritance* techniques. In [1] a protocol is presented that augments exclusive locking with *priority ceiling* support. The priority ceiling protocol limits the *priority inversion* (a lower-priority task blocking a higher-priority task) to at most the time one lower priority transaction holds one lock. This feature facilitates predictability by allowing better schedulability analysis of the entire system [1]. The protocol also prevents deadlock that could result from the locking. In [2] a similar protocol is presented that augments read/write locking with a form of priority ceiling. This protocol also limits priority inversion to the time one lower-priority transaction holds a lock, and it prevents deadlock. In addition, it has the extra advantage of increasing concurrency by using read/write locking instead of the exclusive locking in the original priority ceiling protocol. The increased concurrency is important for faster execution in real-time databases.

In this paper we explore the next step in allowing still more concurrency when priority ceiling techniques are used in *real-time object-oriented databases.* We show this by describing how priority ceiling specification and enforement can be added to *object-based real-time semantic concurrency control* techniques [3]. The resulting protocol provides more potential concurrency for real-time object-oriented databases than previous priority ceiling techniques and alleviates priority inversion and deadlock problems of previous object-based semantic concurrency control techniques. It is also a natural extension of priority ceiling techniques to controlling concurrent access to objects in general.

Section 2 outlines our previous work on object-based semantic real-time concurrency control. It describes our model of a real-time object-oriented database, our semantic locking concurrency control technique, and the priority inversion and deadlock problems that it can have. Section 3 shows how priority ceilings can be added to a restricted version of the semantic locking technique that is based on *affected sets* [4]. The resulting protocol is called the *Affected Set Priority Ceiling Protocol.* Several examples in Section 3 illustrate the relation of our new protocol to ex-

* This work has been sponsored by: The MITRE Corporation, The U.S National Science Foundation, The U.S. Office of Naval Research and the U.S Navy Research and Development Labs (NRaD) Distributed Hybrid Database Architecture project.

isting priority ceiling protocols. Section 4 summarizes and indicates areas where further work is needed.

2 Object-Based Semantic Real-time Concurrency Control

For the past three years our research group at the University of Rhode Island has been performing research in real-time object-oriented databases. This work has included specification of the RTSORAC model [5] for real-time object-oriented databases, and the specification, implementation, and analysis of an associated semantic locking technique for concurrency control [3, 6]. The semantic locking technique allows the designer of individual objects to determine the allowable level of concurrency within an object, based on the semantics of the object. These semantics may require the relaxation of serializability. A critical issue in the field of real-time databases involves the conflicting requirements of logical and temporal consistency. In order to maintain the logical consistency of the data and/or transactions, transactions may be blocked and miss their deadlines, or they may not be able to write data within the data's timing constraints. On the other hand, by allowing a transaction to preempt a conflicting transaction in order to write time-constrained data, the logical consistency of the data or of the transactions may be compromised. The semantic locking technique allows the object designer to explicitly express this trade-off between logical and temporal consistency.

2.1 RTSORAC Model.

The RTSORAC model [5] incorporates features that support the requirements of a real-time database into an extended object-oriented model. It has three components that model the properties of a real-time object-oriented database: *objects*, which represent database entities, *relationships* which represent aggregations of objects, and *transactions*, which represent executable entities that invoke operations on the objects.

The RTSORAC model extends the traditional object-oriented notion of an object to include attributes that have a value, a timestamp and an amount of accumulated imprecision. The imprecision that is recorded accumulates due to the relaxation of serializability by the semantic locking concurrency control technique [6, 7]. RTSORAC objects also include constraints and a compatibility function. The constraints can be placed on the attributes to express logical and temporal correctness of the object.

The user-defined compatibility function determines how the methods of the object may interleave. It is through this function that the object designer expresses the semantics of allowable concurrency. The flexibility of the compatibility function allows the object designer to specify different levels of concurrency for different objects. For instance, one object may require serializability, while another object may tolerate a less restrictive form of correctness. To enforce serializability the object designer may use *affected set semantics* [4] to determine compatibility. A method's *Read Affected Set* (*RA*) is the set of the object's attributes that the method reads. A method's *Write Affected Set* (*WA*) is the set of the object's attributes that the method writes. Under affected set semantics, two methods m_1 and m_2 are compatible if and only if:

$$(WA(m_1) \cap WA(m_2) = \emptyset) \wedge$$
$$(WA(m_1) \cap RA(m_2) = \emptyset) \wedge$$
$$(RA(m_1) \cap WA(m_2) = \emptyset)$$

A less restrictive form of correctness may be needed to express the trade-off between temporal and logical consistency. In such a case, the semantics of compatibility between methods are based on dynamic information, including current temporal consistency and imprecision of data. For example, if a method m_1 that reads an attribute a is currently executing, it would violate the logical consistency of m_1's return value if another method m_2 that writes a were to execute. However, if the timing constraint on a has been violated, i.e. it has become old, then allowing m_2 to execute would restore the temporal consistency of a. When determining each potential allowable interleaving of method executions, the compatibility function can also examine the amount of imprecision that could be introduced by the possible interleaving.

2.2 Semantic Locking Technique.

Our semantic locking concurrency control technique is based on the RTSORAC model and uses *semantic locks* to determine which transactions may invoke methods on an object [3]. The semantic locking technique uses a set of temporal consistency and logical consistency preconditions and the object's compatibility function to determine if a requested semantic lock should be granted.

When a transaction invokes a method m, the technique first tests the logical consistency preconditions to ensure that m does not introduce too much imprecision into the data or the calling transaction. It also tests a temporal consistency precondition to ensure that the m will not read data that would become temporally invalid during its execution. If both preconditions are true, the technique accumulates the imprecision that might be introduced by m, and checks the compatibility function to determine if m is compatible with all other active methods. It also checks that m is compatible with all methods of higher priority that are blocked. If any incompatibilities are found, m is blocked and placed in a priority queue. Otherwise, any imprecision that results from allowing m to interleave with the currently active methods is accumulated, and m is allowed to execute.

We have shown that our semantic locking technique can bound the imprecision that is accumulated due to non-serializable method interleavings. In [6] we proved that under certain general restrictions, the technique maintains a correctness criterion called Object-Oriented Epsilon-Serializability, a specialization of Epsilon-Serializability [7] for the RTSORAC model.

We conducted performance tests in which we compared two versions of our semantic locking technique with other object based concurrency control techniques (exclusive locking, read/write locking and commutative locking) [8]. We looked at the semantic locking technique where temporal consistency was chosen over logical consistency in the trade-off (we refer to it as the *semantic-temporal* technique). We also looked at the semantic locking technique where logical consistency was chosen over temporal consistency in the trade-off (referred to as the *semantic-logical* technique).

We used two measures of performance in our tests. We first measured how many deadlines were missed by each of the techniques. We also measured the amount of data temporal inconsistency seen by each transaction. Our semantic techniques generally met more deadlines than the other object-based techniques. We saw the difference best in cases where methods were short, and transactions had a medium number of method invocations. Furthermore, the semantic techniques generally caused transactions to access less temporally inconsistent data than the traditional techniques.

2.3 Improvements to The Semantic Locking Technique.

Our semantic locking concurrency control technique provides valuable insights into concurrency control in a real-time object-oriented database including: expression of the trade-off between logical and temporal consistency, increased concurrency for meeting more transaction deadlines, and bounding imprecision. However, the original technique suffers from unbounded priority inversion and the possibility of deadlock, both of which can affect the system's predictability and its ability to meet timing constraints.

When a new method is invoked, the technique checks to make sure that the new method invocation is compatible with all blocked method invocations with higher priority. This check avoids a source of priority inversion by ensuring that the new (lower priority) method invocation will not block a waiting higher priority method invocation. However, there are other sources of priority inversion that are not handled by the technique. For instance, assume a transaction $T1$ with priority 1 has invoked a method m_1 that is currently running. If another transaction $T5$ with priority 5 (assume higher number = higher priority) invokes m_2, which is incompatible with m_1, then the higher priority transaction is blocked by the lower priority transaction. Furthermore, medium priority transactions, those having priority between 1 and 5, will be able to execute while $T5$ is blocked because they have higher priority than $T1$. This is a classic instance of unbounded priority inversion. The problem of deadlock is also not addressed by the semantic locking technique.

3 Priority Ceiling in Semantic Locking

To address the problems of priority inversion and deadlock in the semantic locking technique, we use an extension to the priority ceiling protocols [1, 2] that have been proven to limit priority inversion and prevent deadlock. The priority ceiling protocols summarized in Figure 1 differ in the amount of concurrency that they allow. Our protocol, based on affected set semantics, allows more potential concurrency than those of [1, 2] in a real-time object-oriented database. However, the protocol does not allow for the full arbitrary semantics of the semantic locking technique.

In this section we present two previous priority ceiling protocols, along with our affected set priority ceiling protocol. We use an example to demonstrate how our protocol can potentially provide more concurrency than the previous protocols.

Locking	Concurrency	Priority Ceiling Protocol
exclusive	least	Basic Priority Ceiling Protocol [1]
read/write	|	Read/write Priority Ceiling Protocol [2]
r/w affected set [4]	↓	Affected Set Priority Ceiling Protocol (this paper)
semantic	most	(future work)

Figure 1: Locking and Priority Ceiling Techniques

3.1 Previous Priority Ceiling Protocols.

A priority ceiling protocol is based on a major assumption about the system in which it is running. Every object and every transaction in the system must be known a priori in order to gain all of the information needed to execute the protocol. Thus, no dynamic information may be used to determine concurrency control.

There are three basic steps to any of the priority ceiling protocols:

1. Before running, the protocol defines a priority ceiling for each critical section that may be locked. The granularity of these critical sections is the core difference among the various priority ceiling protocols.

2. At run-time, when a transaction $T3$ requests a lock, the lock can be granted only if $T3$'s priority is strictly higher than the ceiling of locks held by all other transactions.

3. If transaction $T3$'s lock request is denied because $T1$ (a lower priority transaction) holds a lock with priority ceiling equal to or greater than $T3$'s priority, $T1$ inherits the priority of $T3$ until $T1$'s lock is released.

Note that no checking of conflict is necessary when granting a lock. This is because, conflict in a priority ceiling protocol is captured in the definition of the priority ceiling.

Each of the protocols that we describe below follow these basic steps. The difference among them arises in how conflict is defined among locks and thus, how priority ceiling is defined. We will describe how priority ceiling is defined in each protocol, and present an example that illustrates how our affected set priority ceiling protocol can provide more concurrency.

The Basic Priority Ceiling Protocol. In the basic priority ceiling protocol [1], exclusive locks are placed on entire objects. Thus, the critical section in this version of the protocol is an object lock. The priority ceiling of a lock is defined as the priority of the highest priority transaction that will ever use this lock. A transaction T can lock a critical section only if it passes the following test:

The priority of transaction T must be strictly higher than the priority ceiling of locks held by all other transactions.

The following example will be used throughout the remainder of this paper. Consider four transactions, $T1$, $T2$, $T3$, and $T4$ in ascending order of priority, where the transaction's subscript represents its priority, sharing two objects OA and OB. For this example, the transactions will execute as follows:

```
T1 : ... lock(OB) ... lock(OA) ...
     release locks ...
T2 : ... lock(OA) ... lock(OB) ...
     release locks ...
T3 : ... lock(OA) ... release lock ...
T4 : ... lock(OA) ... lock(OB) ...
     release locks ...
```

Using this information, the priority ceiling of OA is equal to the priority of $T4$, and the priority ceiling of OB is equal to the priority of $T4$, because $T4$ locks both OA and OB. The following sequence of events represents one possible concurrent interaction of these transactions:

1. Transaction $T1$ locks OB. (*The lock is granted.*)

2. $T2$ preempts $T1$ and attempts to lock OA. (*The priority of $T2$ is not greater than the priority ceiling of OB.*)

3. $T2$ is blocked and $T1$ resumes at priority 2. (*Deadlock is avoided. Priority inversion is limited.*)

4. $T3$ preempts $T1$ and attempts to lock OA. (*The priority of $T3$ is not greater than the priority ceiling of OB.*)

5. $T3$ is blocked and $T1$ resumes at priority 3.(*Priority inversion is limited.*)

6. $T4$ preempts $T1$ and attempts to lock OA. (*The priority of $T4$ is not greater than the priority ceiling of OB.*)

7. $T4$ is blocked and $T1$ resumes at priority 4.

After Event 7), transaction $T1$ continues until it releases all of its locks. At that time $T4$ will be allowed to lock object OA, and complete. The example shows that any transaction with a priority less than $T4$ attempting to acquire a lock will be blocked by the priority ceiling of the lock held by the lower priority transaction $T1$. This means that $T4$ will be blocked as long as $T1$ holds its locks.

One drawback of this protocol for real-time systems is that locking an entire object is very restrictive and can unnecessarily inhibit concurrency that is important to the fast execution that is often needed in real-time databases.

The Read/Write Priority Ceiling Protocol. In a database that allows select, insert, and update functionality, a division can be made between read and write operations. Instead of acquiring an exclusive lock on an entire object, a transaction can request read and write locks. Bounding priority inversion and preventing deadlock with read/write locking has been addressed by the read/write priority ceiling protocol [2].

In the r/w priority ceiling protocol, since each object can allow both readers and writers, each object requires two static priority ceilings, and one dynamic priority ceiling that are defined as follows:

1. The *write priority ceiling* is set equal to the highest priority transaction that will ever write the object.

2. The *absolute priority ceiling* is set equal to the highest priority transaction that will ever read or write the object.

3. The *r/w priority ceiling* is set at run-time. If a transaction is allowed to read an object, the r/w priority ceiling is set equal to the write priority ceiling. If a transaction is allowed to write an object, the r/w priority ceiling is set equal to the absolute priority ceiling.

In the read/write priority ceiling protocol, a critical section is a read/write lock. A transaction T can lock a critical section only if it passes the following test:

The priority of transaction T must be strictly higher than the r/w priority ceiling of locks held by all other transactions.

Object OA	
Abs PC	4
Write PC	3

Object OB	
Abs PC	4
Write PC	2

Figure 2: Example Read/Write Priority Ceilings

Once again, consider the four transactions, $T1$, $T2$, $T3$, and $T4$, sharing two objects OA and OB. However, we can now use the additional information of which transactions are reading and which are writing. The transactions will execute as follows:

```
T1 : ...read_lock(OB)...read_lock(OA)...
     release locks...
T2 : ...write_lock(OA)...write_lock(OB)...
     release locks...
T3 : ...write_lock(OA)...release lock...
T4 : ...read_lock(OA)...read_lock(OB)...
     release locks...
```

Figure 2 depicts the static priority ceilings for each object.

The following is an example of how the transactions might be executed:

1. Transaction $T1$ read_locks OB. (*r/w priority ceiling of OB = write priority ceiling of OB = 2.*)

2. $T2$ preempts $T1$ and attempts to write_lock OA. (*The priority of $T2$ is not greater than the r/w priority ceiling of OB.*)

3. $T2$ is blocked and $T1$ resumes at priority 2. (*Deadlock is avoided. Priority inversion is limited.*)

4. $T3$ preempts $T1$, and attempts to write_lock OA. (*The priority of $T3$ is greater than the r/w priority ceiling of OB = 2.*)

5. $T3$ is granted the write_lock on OA. (*r/w priority ceiling of OA = absolute priority ceiling of OA = 4.*)

6. $T4$ preempts $T3$ and attempts to read_lock OA. (*The priority of $T4$ is not greater than the r/w priority ceiling of OA = 4.*)

7. $T4$ is blocked and $T3$ resumes at priority 4.

The above schedule allows more concurrency than the schedule produced by the basic protocol, while still preventing deadlock. Although this protocol allows more concurrency, it loses effectiveness when individual methods performed on the object can both read

Object OA				
	read_speed	write_speed	read_altitude	write_altitude
read_speed	YES	NO	YES	YES
write_speed	NO	NO	YES	YES
read_altitude	YES	YES	YES	NO
write_altitude	YES	YES	NO	NO

Object OB			
	read_speed	read_depth	write_speed_depth
read_speed	YES	YES	NO
read_depth	YES	YES	NO
write_speed_depth	NO	NO	NO

Figure 3: Example Compatibility Tables

and write an object. If each method writes some piece of information in the object, all locks on the object would be write locks. In this case, this protocol would provide no more concurrency than the basic priority ceiling protocol.

Thus, so far we have seen that the basic priority ceiling protocol works by placing a single ceiling on an entire object, thereby placing an exclusive lock on that object. The r/w priority ceiling protocol places two ceilings on an object, thus possibly allowing many readers to an object at any given time and only one writer. We now continue this pattern in the next section when we define priority ceilings for locking techniques in object-oriented databases.

3.2 Affected Set PC Protocol.

The previous priority ceiling protocols place a priority ceiling on an entire data object and therefore allow less potential concurrency than semantics-based techniques, such as that described in Section 2, that use locks on methods of database objects.

Our *Affected Set Priority Ceiling* (ASPC) protocol [9] uses the affected sets [4] of each method of each object to determine the compatibilities of the methods of an object. Our previous semantic locking technique (Section 2) uses affected set information, but also allows the object designer to specify additional conditions under which methods may execute concurrently. Because priority ceiling protocols are based on static information, establishing priority ceilings where arbitrary semantics are allowed is not straightforward. Thus, the approach in this paper focuses on affected set semantics.

Using affect set semantics, the critical section is a method lock. Thus, the ASPC protocol assigns a *conflict priority ceiling* to each method of each object. The conflict priority ceiling of a method m is the priority of the highest priority transaction that will ever lock a method that is not compatible with method m (based on affected set semantics - see Section 2).

The ASPC protocol allows a transaction T to receive a lock on a critical section if and only if:

The priority of T is strictly higher than the conflict priority ceiling of locks held by all other transactions.

We further expand upon the previous example to illustrate the benefits of our ASPC protocol. Consider two database objects:

```
Object OA :
  Attr  speed;
  Attr  altitude;

  method read_speed();     /*RA = speed */
  method write_speed();    /*WA = speed */
  method read_altitude(); /*RA = altitude */
  method write_altitude(); /*WA = altitude */


Object OB :
  Attr  speed;
  Attr  depth;

  method read_speed(); /*RA = speed */
  method read_depth(); /*RA = depth */
  method write_speed_depth();
/*WA = speed, depth */
```

For simplicity these objects were defined to have distinct read and write methods; however, methods are not restricted to this behavior. Notice that object OA has separate methods to write each attribute (Attr), while OB has a method that writes to two attributes.

Object OA				
method →	read_speed	write_speed	read_altitude	write_altitude
Highest Priority Transaction	T1	T3	T4	T3
Conflict Priority Ceiling	3	3	3	4

Object OB			
method →	read_speed	read_depth	write_speed_depth
Highest Priority Transaction	T1	T4	T2
Conflict Priority Ceiling	2	2	4

Figure 4: Example Affected Set Priority Ceilings

Each object is analyzed to determine the read/write affected set for each of its methods, as shown by the `RA` and `WA` annotations. Figure 3 (previous page) displays the method compatibilities for objects OA and OB, based on affected set semantics.

Next, the protocol examines the transactions that access the methods. Assume the following transactions:

```
T1 :...method_lock(OB, read_speed)...
    method_lock(OA, read_speed)...
    release locks
T2 :...method_lock(OA, write_speed)...
    method_lock(OB, write_speed_depth)...
    release locks
T3 :... method_lock(OA, write_speed)...
    method_lock(OA, write_altitude)...
    release locks
T4 :...method_lock(OA, read_altitude)...
    method_lock(OB, read_depth)...
    release lock
```

The conflict priority ceilings for the methods in our example are displayed in Figure 4.

The following represents one possible concurrent execution of these transactions.

1. Transaction $T1$ method_lock(OB, read_speed). (*The lock is granted.*)
2. $T2$ preempts $T1$ and attempts method_lock(OA, write_speed). (*The priority of $T2$ is not greater than the conflict priority ceiling of OB, method read_speed.*)
3. $T2$ is blocked and $T1$ resumes and inherits priority 2. (*Deadlock is avoided.*)
4. $T3$ preempts $T1$ and attempts method_lock(OA, write_speed). (*The priority of $T3$ is greater than the conflict priority ceiling of OB, method read_speed.*)
5. $T3$ is granted the method_lock.
6. $T4$ preempts $T3$ and attempts method_lock(OA, read_altitude). (*The priority of $T4$ is greater than the conflict priority ceilings of OB, method read_speed and OA, method write_speed.*)
7. $T4$ is granted the lock and continues to completion.

Once $T4$ completes, $T3$ will resume and eventually release its locks. After $T3$ completes, $T1$ will resume and after releasing the *method_lock*(OB, *read_speed*), will revert back to priority 1. This allows $T2$ to preempt and run to completion. Finally, $T1$ will complete. Note that in this example, the ASPC protocol allows two more locks to be granted than the basic protocol and one more than the read/write protocol. Furthermore, the blocking time for the high priority transaction, $T4$, is reduced. In the example of the basic protocol, $T4$ is blocked as long as the lower priority transaction $T1$ holds its locks. With the ASPC protocol, $T4$ is not blocked at all.

This example provides intuition for the effectiveness of the ASPC protocol. It indicates that the finer granularity ceilings can provide more concurrency than the other protocols, and that blocking time for high priority transactions can be reduced. This possible reduction in blocking time is the result of the potentially shorter critical sections (method locks vs. object or read/write locks) of the ASPC protocol. A formal proof for deadlock prevention and bounding priority inversion in the previous priority ceiling protocols appear in [1, 2]. We are developing similar proofs showing deadlock prevention and priority inversion bounds in the ASPC protocol [9].

4 Conclusion

The ASPC protocol incorporates priority inversion bounds and deadlock prevention into the semantic locking technique with the semantics restricted to affected sets. It is also an important step towards applying priority ceiling techniques to real-time object-oriented databases. Furthermore, the generality of the ASPC protocol makes it a natural step in extending priority ceiling techniques to control concurrent access to objects.

There are several drawbacks to the ASPC protocol for real-time databases. First, like all priority ceiling techniques, it requires a substantial amount of static, *a priori* information about the system. It is this extra static information that allows priority inversion bounding and deadlock prevention, but it can be a prohibitive assumption for dynamic real-time systems. Second, the ASPC protocol enforces serializability of methods, which may be overly restrictive for real-time databases. Finally, the ASPC protocol does not address temporal consistency nor does it explicitly handle the trade-off between temporal and logical consistency.

The extension of the ASPC protocol to allow the full semantics of the semantic locking technique, which can address each those drawbacks, is complicated. The fundamental problem is that priority ceiling protocols rely on static knowledge to determine ceilings, while full semantic conditions, such as current temporal consistency status, are dynamic in nature. The ASPC protocol is a compromise that allows more concurrency than previous priority ceiling techniques, and solves the deadlock and priority inversion problems of the semantic locking technique. To bound priority inversion and prevent deadlock in full semantic locking, further research is required.

References

[1] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, pp. 1175–1185, Sept. 1990.

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

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

[5] 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.

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

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

[8] L. C. DiPippo, *Object-Based Semantic Real-Time Concurrency Control*. PhD thesis, Department of Computer Science, The University of Rhode Island, 1995.

[9] M. Squadrito, "Extending the priority ceiling protocol using read/write affected sets," Master's thesis, Department of Computer Science, The University of Rhode Island, 1995.