OBJECT-BASED SEMANTIC REAL-TIME CONCURRENCY CONTROL

BY

LISA B. DIPIPPO

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

APPLIED MATHEMATICAL SCIENCE

UNIVERSITY OF RHODE ISLAND

1995

DOCTOR OF PHILOSOPHY DISSERTATION

OF

LISA B. DIPIPPO

APPROVED:
Dissertation Committee
Major Professor_____

_____

_____

_____
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
1995

ABSTRACT

Concurrency control for a real-time database must maintain both the traditional logical consistency constraints of data and transactions, and the additional temporal consistency constraints of data and transactions. Furthermore, the concurrency control should have the ability to express the trade-off that results from the inherent conflict between temporal and logical consistency constraints. The concurrency control should also be able to maintain and bound any imprecision that results from trading off logical consistency for temporal consistency.

This dissertation presents a model and a concurrency control technique for real-time object-oriented databases. The model, called RTSORAC, is based on an object-oriented data model and it allows for the explicit expression of logical and temporal data consistency constraints. The concurrency control technique supports temporal and logical consistency, as well as bounded imprecision that results from their trade-off. It uses a *semantic locking* mechanism within each object and user-defined compatibility over the methods of the object. The semantic-based compatibility function can specify when to sacrifice precise logical consistency to meet temporal consistency requirements. The concurrency control technique can also specify accumulation and bounding of any resulting logical imprecision.

The dissertation presents a set of restrictions on the compatibility function and an object-oriented form of the epsilon-serializability correctness criterion (OESR). It then presents a proof of global correctness that shows that the semantic locking technique, under the compatibility function restrictions, can guarantee OESR.

To demonstrate how the semantic locking technique performs under varying conditions, the dissertation presents a set of tests that were conducted to compare the technique with other techniques. The techniques indicate that, in general, the semantic locking technique preserves transaction temporal consistency better than the other techniques tested, and it keeps data temporal inconsistency low.

ACKNOWLEDGEMENT

I have always thought of the work in this dissertation as a collaborative effort between me and so many people who have helped me in many ways, not just academically.

First I must thank my advisor and friend Victor Fay-Wolfe, without whom I would not have known what a real-time database was. He has been an advisor in more than just my work and has provided me with all of the support a graduate student could ask for - and then some.

I also thank Joan Peckham, who has been a role model for me in work and in life and in combining the two! My fellow graduate student Janet Prichard has been a supportive friend as well as a helpful colleague. I thank the entire RTSORAC research group for all of the valuable input I have gotten throughout the process of my research.

I thank Manbir Sodhi and the rest of my committee for their support and help throughout the entire PhD process. Many thanks to John Kelvin Black, who provided me with a a solid testbed system upon which to design and run my performance tests. I also have to thank Roman Ginis for all of the little tasks that he performed in order to help me get my tests running. I thank Colleen Kelly for her help in analyzing the results of my testing. I must also give special thanks to *elvis* for working with me night and day to get my testing done.

Finally, I must thank my family for all of the love and support that they have given me throughout the years. And for all of the encouragement that I ever got that enabled me to get this far. I thank my husband John for all of his love, and for sticking with me from start to finish in this endeavor. And last, but definitely not least, I thank my little Kaiya Lee for sleeping when I needed her to, and more importantly for giving me the perspective I needed in school, in work and in life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Applications such as submarine contact tracking, program stock trading, and medical patient monitoring require timely actions as well as the ability to access and store complex data that reflects the state of the application's environment. A traditional database provides some of the functionality required by these applications, such as coordination of concurrent actions and consistent access to shared data. *Real-time databases* have all of the features of traditional databases, but they can also express and maintain *time-constrained data* and *time-constrained transactions* [Ram93]. *Object-oriented databases* provide a mechanism for expressing and accessing complex data in an encapsulated form with a well-defined interface [ZM90]. These added features of real-time databases and object-oriented databases present more complexity than exists in traditional databases and this complexity affects how the database controls access to its data.

This dissertation defines concurrency control requirements of a real-time object-oriented database and describes a model and a concurrency control technique that attempt to meet these requirements. It presents a new correctness criterion and a proof of correctness of the technique. Finally, the dissertation describes a prototype implementation on which performance tests were conducted, and analyzes the results.

1.1 Motivation

A *transaction* is an executable entity that is the unit of concurrency in a database. The *concurrency control* mechanism of a traditional database coordinates actions of transactions that operate concurrently and access shared data [BHG86]. Determining which transactions

may execute concurrently depends on the *correctness criterion* chosen for the concurrency control mechanism. One such correctness criterion is *mutual exclusion* which requires that each transaction execute from start to finish without interruption from other transactions, thus no concurrency is allowed. Most traditional concurrency control techniques require *serializability* of its transactions. A schedule of transactions is considered to be *serializable* if it produces the same output as some serial execution of the same transactions [BHG86]. Concurrency control mechanisms that require mutual exclusion or serializability usually ensure that each transaction that accesses shared data remains *atomic*, either all of its actions are performed or none are, and *exclusive*, it does not interfere with other transactions [BHG86]. These requirements place constraints on how the transactions may execute (*transaction logical consistency* constraints) and on how the data may be accessed (*data logical consistency* constraints).

The concurrency control mechanism of a real-time database must maintain not only the logical consistency of the database, it must also maintain *transaction temporal consistency* and *data temporal consistency*. Transaction temporal consistency constrains when a transaction must execute by imposing timing constraints such as deadlines, earliest start times and latest start times. Often transactions are assigned *priorities* based on these timing constraints. The real-time scheduler must use these priorities when scheduling transactions on the CPU. Data temporal consistency constrains how old a data item may be and still be considered valid. The concurrency control mechanism for a real-time database must maintain all four forms of consistency constraint: transaction logical consistency, data logical consistency, transaction temporal consistency and data temporal consistency.

Unfortunately, there exist certain inherent conflicts between temporal and logical constraints that make it difficult for a concurrency control mechanism to maintain all four forms of constraints. For instance, in order to maintain transaction temporal consistency, a transaction that updates a piece of data may be blocked by another transaction that is reading the same piece of data. If the data in question is getting "old" it would be in the interest of its temporal consistency to allow the updating transaction to execute. However, this execution could violate the logical consistency of the data or of the reading transaction. Thus, there is a trade-off between maintaining temporal consistency and maintaining logical consistency. If logical consistency is chosen, then there is the possibility that a piece of data may become old, or that a transaction may violate a timing constraint. If, on the other

hand, temporal consistency is chosen, the consistency of the data or of the transactions involved may be compromised.

Another problem that may occur due to the conflict between logical and temporal constraints is called *priority inversion*. This occurs when a transaction is blocked by a lower priority transaction because the lower priority transaction holds locks on data that the higher priority transaction is waiting for. The temporal consistency constraints of the transactions impose the priority ordering, but the logical consistency constraints of the transactions forces the higher priority transaction to be blocked by the lower priority transaction holding the lock.

In order to express the trade-off between temporal and logical consistency that exists in a real-time database, a concurrency control mechanism must be able to use knowledge about the database application in determining which transactions may execute concurrently. Such a mechanism is said to provide *semantic concurrency control*. By using semantic information about the database and the particular application domain, the database designer can define less restrictive correctness criteria, sometimes relaxing the requirement for serializability.

A consequence of relaxing serializability is that *imprecision* may accumulate in the data in the database and in the transactions' views of the data. Recall the example described earlier in which an update transaction preempts a reading transaction in order to maintain the temporal consistency of the data. In this case, the reading transaction may get an imprecise view of the data because it may read the value written by an uncommitted update transaction. A data value resulting from a schedule of transactions is *imprecise* if it differs from the corresponding value resulting from each possible serializable schedule of the same transactions [RP]. The imprecision of a data item may be local to the view of a single transaction, such as when one transaction reads data written by another uncommitted transaction. A data item may also be imprecise with respect to future transactions that access it, such as when two transactions that write to the data item interleave.

Semantic concurrency control can facilitate the expression of the trade-off between temporal and logical consistency, and therefore help to maintain the temporal consistency of the data. It can also help to maintain the temporal consistency of the transactions. Figure 1.1 illustrates how the choice of concurrency control correctness criteria can affect the database's ability to maintain transaction temporal consistency. It shows that the set of schedules that produce serial execution of transactions (using mutual exclusion, for exam-

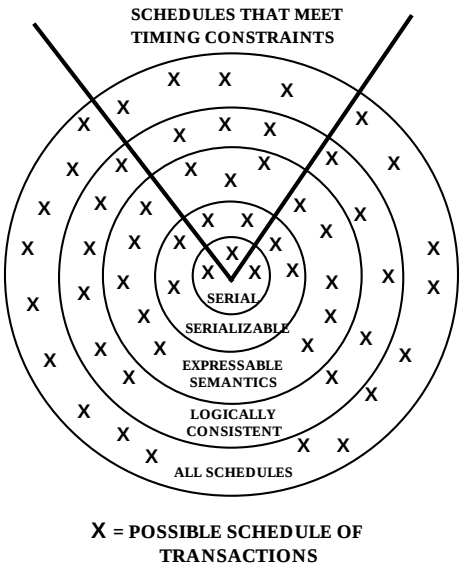

Figure 1.1: Concurrency Control Allowable Schedules

ple) is a very small subset of logically consistent schedules. The set of serializable schedules is also a small subset. However, when semantics of the application are used to determine concurrency, a larger subset of logically consistent schedules results. Allowing as large a subset as possible is important in a real-time database because the more logically consistent schedules that the correctness criterion allows, the more flexibility the scheduler has in finding a schedule that is both logically consistent and temporally consistent.

Most traditional databases are based upon the relational data model in which data is stored in a set of tables [Dat86]. An object-oriented data model supports functionality beyond that of the relational data model [ZM90]. Data in an object-oriented data model is represented by *objects*, each with its own unique *object identity*. *Encapsulation* of data and implementation provides a well-defined interface to each object through operations called *methods*. While the representation of the data or the implementation may change, the interface remains the same and thus, users of the object are unaware of the changes. An object-oriented data model supports objects with *complex state*, that is, objects that may refer to other objects.

While most work in concurrency control for real-time databases has used the relational data model [YWLS94], there are several reasons that an object-oriented data model [ZM90] can be more suitable for the complex real-time applications described above. First, the encapsulation mechanisms of an object-oriented data model facilitate expressing semantic

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

Table 1.1: Forms of Consistency for Real-Time Database

concurrency control information within each object. Because the concurrency control is defined by the object designer, the trade-off between temporal and logical consistency can be expressed based on the semantics of the object. Second, the capability to include user-defined methods on data objects can improve real-time concurrency by allowing a wide range of operation granularities for semantic concurrency control. Finally, constraints can be expressed as first-class entities in an object-oriented data model, thus allowing the expression of temporal, logical and imprecision constraints as an integral part of the data.

1.2 Goal of Research

Our goal is to provide a concurrency control technique for a real-time object-oriented database that supports all four forms of consistency constraints (Table 1.1) and the trade-offs that result.

1.3 Our Approach

In order to meet our goal, we have first defined a model for real-time object-oriented databases, called RTSORAC (**R**eal **T**ime **S**emantic **O**bjects, **R**elationships **A**nd **C**onstraints) [PDPW94]. The model incorporates functionality of traditional databases with the requirements of real-time databases and the additional features of object-oriented databases. We have devised a concurrency control technique based on the RTSORAC model, called *semantic locking*. In our 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 its methods.

Semantic locking supports data and transaction logical consistency by providing functionality similar to concurrency control techniques for non real-time databases. It supports data temporal consistency by allowing for the expression and enforcement of the trade-off of logical consistency for temporal consistency. The technique supports transaction tempo-

ral consistency by allowing for the specification of more logically consistent schedules than those allowed by serializability (Figure 1.1). Furthermore, semantic locking can specify accumulation and bounding of any logical imprecision that may result from relaxing serializability. We have proven that our technique can bound imprecision by showing that it can guarantee a form of epsilon serializability [RP] specialized for object-oriented databases.

1.4 Dissertation Outline

In Chapter 2 we review related work and evaluate it to indicate why each does not sufficiently support the four forms of consistency of Table 1.1 and the associated trade-offs. Chapter 3 defines our RTSORAC model for real-time object-oriented databases. Chapter 4 describes our semantic locking technique. In Chapter 5 we present a new correctness criterion and use it to prove that semantic locking can be globally correct and that it can bound imprecision. Chapter 6 describes the prototype implementation of the RTSORAC model and of our technique. Chapter 7 presents the results of performance tests using simulated workloads. The tests indicate how our technique supports the transaction and data temporal consistency. Chapter 8 compares our work with related work, presents the contributions and limitations of our work, and discusses future work.

Chapter 2

Related Work

In this chapter we describe some of the related work that has been done in the area of concurrency control. We discuss several techniques in order of increasing semantics used. The first techniques that we present require mutual exclusion in which no semantics are used. The traditional serializability techniques that we describe next use the semantics of read and write operations to determine how much concurrency to allow. We present semantic concurrency control techniques in which the semantics of the actual applications are used to further enhance the amount of concurrency used. We also describe several correctness criteria and techniques that allow bounded imprecision. The chapter ends with an evaluation of the related work based on its ability to support the four consistency requirements: transaction logical consistency, data logical consistency, transaction temporal consistency and data temporal consistency.

2.1 Mutual Exclusion Techniques

Mutual exclusion requires that only a single transaction or process can access a resource at a time. It does not allow any concurrency among transactions. The programming language Ada [DoD83] uses *rendezvous* to enforce mutual exclusion. The technique is called rendezvous because a *caller* task must wait for an accept from a *server* task and the server must wait at an accept for a call from a caller. The two tasks meet at this rendezvous point and while the server is processing the request from the caller, no other caller may rendezvous with the server.

Real-Time Euclid [KS86], a third generation language designed for real-time applica-

tions, uses *monitors* to maintain mutual exclusion. A monitor is an abstract data type that contains both the data and procedures needed to perform allocation of a particular shared resource or group of shared resources [Dei84]. A process calls a monitor entry and mutual exclusion is strictly enforced at the monitor boundary. Real-Time Euclid has extended the standard first-in first-out wait queue to handle time. A time bound is specified on the wait statement so that if time runs out on a process that is waiting for a monitor entry an exception is raised that will handle the missed timing constraint.

2.2 Traditional Serializability Techniques

Some concurrency control techniques that require serializability use certain semantic knowledge of the application to gain extra concurrency. We describe only those serializability techniques that use no application-specific knowledge and describe semantic based techniques in the next subsection.

Read/write locking is a technique in which two types of locks are provided for each data item based upon the operation that is requested. A read lock is compatible with other read locks, but a write lock is not compatible with read locks or other write locks. When a lock is requested on a data item, it is granted if it is compatible with all other locks already held on the particular data item and denied otherwise. Read/write locking is used to maintain serializability of the operations on a data item [BHG86].

Two-phase locking [BHG86] is a concurrency control technique that is used to maintain the serializability of transactions. It is used along with some type of compatibility locking technique like read/write locking. Two-phase locking requires that a transaction acquire all of the locks that it needs before releasing any locks. Therefore, it cannot release a lock until all locks are acquired and it cannot request a lock once another lock has been released.

Certain variations of two-phase locking have been developed for particular purposes [BHG86]. For instance, conservative two-phase locking requires that all locks are acquired by a transaction before any other processing is done. This is done to avoid deadlock. Strict two-phase locking forces all transactions to hold their locks until they are terminated either by a commit or an abort.

There are several real-time concurrency control techniques that extend two-phase locking to take into account real-time requirements. In [SRSC91], the *priority ceiling protocol*

[Raj89] real-time scheduling algorithm is combined with a version of two-phase locking to handle the priority inversion problem. Using this protocol, priority inversion is bounded so that a transaction can be blocked by at most one lower priority transaction until it completes or suspends itself.

In [AGM88], the *2PL-HP* (two-phase locking with high priority) protocol was proposed. In this protocol, conflicts are resolved by aborting lower priority transactions. If a transaction requesting access to shared data has a higher priority than all other transactions holding locks on the data, the lock holders abort and the requester gets the lock. Otherwise the requester waits for the holder to release the lock. Another variation of this idea, called *H2PL* (hybrid 2PL), was proposed in [HL92]. In this technique certain conditions, such as transaction workload, are checked to avoid unnecessary aborts. Also, whenever a lower priority transaction that is blocking a higher priority transaction aborts and therefore has to be restarted, its priority is raised to that of the higher transaction to prevent priority inversion (*priority inheritence*).

Timestamp ordering is a serializability technique that does not use locks, but rather determines which interleavings to allow based upon the time at which the transactions were initiated [BHG86]. Each transaction receives a timestamp. Given two concurrent transactions, T_1 and T_2, if two conflicting operations within these transactions, o_1 and o_2, access shared data, then o_1 is processed before o_2 if and only if the timestamp of T_1 is before the timestamp of T_2. The concurrency control mechanism receives operations on a first come first served basis, but rejects any operations that it receives "too late". That is, if o_1 is received and it conflicts with o_2 and the timestamp of T_1 is before the timestamp of T_2, but o_2 was received first, o_1 is too late. It is rejected and therefore T_1 is aborted.

All of the serializability techniques described above are pessimistic in nature. That is, a conflict is detected when it occurs and some action is taken right away. For instance, in two-phase locking and its variations, when a lock conflict is detected, one of the conflicting transactions is either blocked or aborted. Similarly, with timestamp ordering, conflicting transactions are handled at the time the conflict occurs. *Optimistic concurrency control* [KR81] techniques detect conflicts after the data access occurs. Each transaction completes its execution, including all data accesses, assuming that no other conflicting actions are executing concurrently. When a transaction completes its execution, it enters a validation phase in which all data accesses are validated. If the transaction did not perform any

actions that conflict with the actions of an already committed transaction, the concurrency control manager marks any uncommitted conflicting transaction for abort. Otherwise, if the transaction has been marked for abort, it is aborted. Optimistic concurrency control can use either locks or timestamps for detection of conflicts.

A study of real-time concurrency control techniques in [HCL90b] indicates that in systems in which late transactions are discarded, a real-time optimistic concurrency control mechanism outperforms the pessimistic technique of [AGM88]. In [HCL90a], a real-time optimistic concurrency protocol called WAIT-50 is presented. In this protocol, a lower priority transaction waits at validation time for any conflicting higher priority transactions to give the higher priority transactions a chance to meet their deadlines first. A wait control mechanism monitors transaction conflict states and dynamically decides when and how long a low priority transaction should wait for its conflicting higher priority transactions.

A real-time optimistic concurrency control technique called OCC-TI [LS93] uses timestamp intervals to detect conflicts. Every transaction is assigned an initial timestamp interval of $[0, \infty)$. The interval is adjusted to represent serialization order dependencies. A final timestamp is assigned from the interval at the end of the validation phase. The validation of a transaction consists of adjusting timestamp intervals of concurrent transactions and restarting conflicting transactions whose intervals cannot be adjusted. This technique uses the concept of dynamic adjustment of serialization order presented in [LS90].

2.3 Semantic Concurrency Control Techniques

A semantic concurrency control mechanism utilizes application specific knowledge to increase concurrency, sometimes defining less restrictive correctness criteria than serializability. Most of the previous work in semantic concurrency control can be divided into two categories: *transaction-based semantic concurrency control* and *object-based semantic concurrency control*. Transaction-based semantic concurrency control capitalizes on the semantics of the known transactions in the system to allow interleavings that might not be allowed in a traditional scheme. Object-based semantic concurrency control manages access to each object in the system based on the semantics of the operations defined on the object. The remainder of this section briefly discusses some of the previous work done in each of these categories of semantic concurrency control.

2.3.1 Transaction-Based Semantic Concurrency Control

In [GM83], Garcia-Molina defines a *semantically consistent schedule* to be a schedule that transforms the database into a consistent state. Transactions are classified into semantic types based on what they do in the database. For each type, a *compatibility set* is defined to identify which other types are compatible with, i.e., may interleave with, the given type. The user divides a transaction type into *atomic steps* where a step represents some indivisible, real-world action. Any interleaving that is allowed is between these user-defined steps. In the transaction processing mechanism proposed, when a transaction requires access to a data object, a lock is requested. If no other locks are held on the object, the request is granted and the object keeps track of the compatibility set of the type of transaction holding the lock. If another transaction attempts to lock the same object, the transaction processing mechanism checks to see if the type of the requesting transaction is in the compatibility set of the transaction already holding the lock. If so, the lock is granted, if not, the transaction must wait to gain access to the object. In this technique, serializability is replaced as a correctness criterion by *semantic consistency*.

In [CGM85] a performance evaluation is done to identify the conditions under which it may be advantageous to use an application-dependent concurrency control mechanism such as the one described in [GM83]. The tests in [CGM85] determined that probability of saved conflict is an important factor for identifying when to use a semantic concurrency control technique. That is, they found that in an application where semantic compatibilities allow the system to avoid more conflicts than traditional, non-semantic-based techniques like two-phase locking, it is beneficial to use these semantic techniques. The evaluation also indicated that if a database is small or has frequently accessed portions and the number of concurrent transactions is high, an application dependent concurrency control mechanism could be useful.

In [Lyn83] an approach similar to [GM83] is proposed. In [Lyn83], each transaction has a different set of breakpoints with respect to each different transaction type. This approach allows varying levels of concurrency among different types of transactions. Transactions are grouped into *nested classes*. As the classes become more refined, the level of atomicity becomes finer. For each class, breakpoints are inserted in a transaction which define places where other transactions of the same class may interleave. The breakpoints of higher level classes are carried down to the lower level classes. Therefore, for each transaction t, the

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

Table 2.1: Lock Compatibility Table

set of breakpoints where another transaction t' can interrupt is determined by the lowest class containing both t and t'. The levels of atomicity produced by this technique form a hierarchy of allowable interleavings among transactions.

Another transaction-based semantic concurrency control mechanism is described in [FO89]. This work extends the previous work described in [GM83] and [Lyn83] by creating fewer restrictions on allowable interleavings. Nested classes are not used, and therefore the interleavings are not required to be hierarchical as in [Lyn83]. Transactions are classified by types and are divided by placing breakpoints between operations where certain interleavings are allowed. Each breakpoint has associated with it a set, called the *interleaving set*, containing the types of transactions which are permitted to interrupt at that point. Four kinds of locks are used in the concurrency control technique described: *shared, exclusive, relatively shared* and *relatively exclusive*. A shared lock or exclusive lock is granted in the traditional way for read access or write access respectively. Relatively shared and relatively exclusive locks are used to produce non-serializable interleavings. At a breakpoint, the lock can change depending on the actions taken before that point. A shared lock becomes a relatively shared lock at a breakpoint if there is no update before it, otherwise it becomes an exclusive lock. An exclusive lock always becomes a relatively exclusive lock at a breakpoint. A compatibility table, as seen in Table 2.1, is given for these four locks and while some of the entries are simply YES or NO, others, labelled COND, depend on whether or not the type of the transaction requesting the lock is in the interleaving set of the type of the transaction holding the lock. Locks are released after termination of the transaction.

The RTC language [WDL93], which also has object-based features, provides mechanisms for concurrency among transactions. For instance, an **exclusive** block ensures that any execution within the block is exclusive of interruption from any other incompatible actions.

The work in [ABAK94], generalizes the previous work in transaction based semantic concurrency control and presents a formal method for determining correct schedules. An

atomic unit of a transaction T_i relative to another transaction T_j is defined to be a sequence of consecutive operations of T_i such that no operations of T_j are allowed to be executed within this sequence. $Atomicity(T_i, T_j)$ refers to the ordered sequence of atomic units of T_i relative to T_j. A schedule of transactions is a *relatively atomic schedule* if for all transactions T_i and T_j, no operation of T_i is interleaved with an atomic unit of T_j relative to T_i.

The authors of [ABAK94] recognize that in general, relative atomicity specifications tend to be conservative because not all potential conflicts occur. They expand the class of relatively atomic schedules to include interleavings of operations which do not have any dependencies between them. An operation o_2 directly depends on an operation o_1 if o_1 precedes o_2 and either both operations are in the same transaction or o_1 conflicts with o_2. A *relatively serial schedule* is defined to be analogous to the notion of serial schedules in the serializability theory. A schedule is relatively serial if for all transactions T_i and T_j, if an operation o of T_i is interleaved with an atomic unit U of T_j relative to T_i, then o does not depend on any operation p in U, and any other operation q in U does not depend on o. A schedule is *relatively serializable* if it is conflict equivalent to some relatively serial schedule. This definition provides a new correctness criterion for transaction based concurrency control. Further, the authors present a method for determining if a given schedule is relatively serial by testing for acyclicity of a directed graph.

2.3.2 Object-Based Semantic Concurrency Control

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

In [BR88], an object-based semantic concurrency control technique is used in a system which allows nested data objects, i.e., objects containing other objects. A hierarchical structure, called a *granularity graph*, is used to represent the nesting. The outermost object is represented at the root of the graph and the children of the root represent the objects nested inside. For each operation defined on the object, an *affected set* is computed, containing all nodes in the graph that are affected by the operation. Concurrency is controlled by avoiding conflicts among the operations on the object. A conflict occurs between two operations if they do not *commute*, that is, if the order in which they are performed affects the results returned by the operations or the resulting state of the object. The approach to determining compatibilities between operations is divided into two steps. First, the se-

mantics of the operations are analyzed to determine if they are always compatible, never compatible or conditionally compatible. The second step is performed dynamically when the operations are requested, to determine the value of a conditional compatibility. This value is determined by computing the intersection of the affected sets of the two operations in question. If this intersection is empty, then the operations commute and therefore are compatible.

Another object-based mechanism that uses commutativity as the definition of compatibility is described in [Wei88]. Two slightly different versions of commutativity are defined, *forward commutativity* and *backward commutativity*. The difference between these criteria is subtle and the author asserts that they are both necessary because each one is used with different recovery mechanisms. Forward commutativity is designed to work with intentions lists, while backward commutativity works with recovery using undo logs. One of the major results of [Wei88] is that concurrency control and recovery are closely linked and must be considered together. When compatibility between operations is in question, commutativity is computed dynamically, as in [BR88].

In [BR92] another technique very similar to that in [BR88] is described. However, in the former, compatibility between operations is based on *recoverability* and not on commutativity. An operation, o_1 is recoverable relative to another operation, o_2, if the outcome of performing o_2 is the same whether or not o_1 executed immediately before o_2. Therefore, recoverable operations are allowed to execute concurrently but must commit in the order in which they were invoked.

The three object-based semantic concurrency control techniques described above add concurrency to a database by exploiting the semantics of the object's operations, but each ultimately requires serializability as a correctness criterion. Other researchers have increased concurrency even further by relaxing the serializability constraint. In both [WDL93] and [SS84] the database designer defines the compatibility between operations on an object. This user-defined compatibility may or may not preserve serializability. Consistency constraints are determined by the designer and implemented through the compatibility relations.

In [WDL93], RTC is a language proposed to control real-time concurrency. Objects called *resources* have actions defined on them. The *compatibility relation* C_r is a non-symmetric relation on these actions which determines if two actions are compatible, that is, if the actions can be overlapped to result in a consistent state of the resource. The designer

of the system must ensure the correctness of the compatibility relation with respect to the semantics of the resource being defined.

In [SS84] as well, the user is responsible for defining compatibilities, but the authors present some guidelines for doing so. The user defines all possible dependencies among the operations of an object, possibly involving values of parameters. Some of these dependencies are characterized as insignificant in that cycles formed by them do not affect data consistency. Rather than using serializability as the correctness criteria, a schedule is considered correct if it is orderable with respect to a relation formed by combining all of the significant dependencies in the objects involved.

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

2.4 Bounded Imprecision

In many real-time systems, imprecise results have been considered acceptable to allow timing constraints to be met [LLS+91]. Some of the semantic concurrency control techniques described above allow imprecision in an ad hoc way in order to provide more concurrency [GM83, Lyn83, SS84]. Here we describe several correctness criteria that formalize the concept of imprecision along with several concurrency control techniques that use these criteria.

In [KM92] the use of imprecision in databases and in real-time systems is synthesized and formalized through the concept of *similarity*. The authors define new correctness criteria, less restrictive than serializability, based on the idea that data values that are sufficiently close may be interchanged as input to a transaction without undue adverse effects.

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

The *Similarity Stack Protocol* (SSP) described in [KM93] defines similarity of data based on the time at which the data is written. Two data items are considered to be similar if their timestamps are within a specified bound. Transactions are placed on a scheduling stack according to their priorities. Read/write events of different transactions may swap positions on the stack as long as they are similar.

Epsilon Serializability. Epsilon serializability (ESR) [RP, DP93] 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. In order to accumulate and limit imprecision, ESR assumes use of only data items that 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 [RP].

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 [RP]. 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 [DP93].

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 [RP] as follows:[1]

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

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

Data safety is described informally in [DP93]. 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 [RP, DP93] 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.

Several concurrency control techniques have been designed to maintain ESR instead of the more restrictive serializability criterion. [WYP92] describes several concurrency control techniques in which read-only transaction need not be serializable with other update transactions, but update transactions must be serializable among themselves. The techniques are variations of two-phase locking, timestamp ordering and optimistic concurrency control. The concurrency control protocols presented in [PHK+93] extend the notion of epsilon serializability to distributed databases. They allow divergence from consistency among database sites as long as their differences remain within specified limits.

[WA92] presents another concurrency control protocol that allows bounded inconsistency. The protocol works on an object-based model. In this model, a transaction invokes an operation on an object and the object has a set of possible actions, called the *resolution set*, from which to execute the operation. The state of an object is defined by the sequence of resolutions that have been performed in response to invoked operations. Two resolution sequences are considered equivalent if the resulting object states are the same.

The object designer determines compatibility of object operations based on the notion of *commutativity* with bounded inconsistency. For every resolution sequence $o_p.o_q$ of the operation sequence $p.q$, the designer defines a *forward resolution set dilating function* (f_{pq}) and a *backward resolution set dilating function* (b_{pq}). These functions are defined such that for every state s, if there is a resolution sequence $o_p.o_q$ of the operation sequence $p.q$ with o_p in the resolution set of p ($rs(p)$) and o_q in the resolution set of q ($rs(q)$), then there exists a resolution sequence $o'_q.o'_p$ that is equivalent to $o_p.o_q$ for the operation sequence $q.p$ with

o'_p in $f_{pq}(rs(p))$ and o'_q in $b_{pq}(rs(q))$.

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

2.5 Evaluation of Related Work

In this section we evaluate the related work that we have described based on our goals for concurrency control in real-time, object-oriented databases. We do this in order to determine which of the previously described techniques can help us in our own work and to identify important issues that have not been addressed in this area.

2.5.1 Transaction Temporal Consistency

Real-Time Euclid was specifically designed to handle real-time applications and therefore it is well equipped to handle temporal constraints on transactions. It allows both periodic and aperiodic transactions as well as timing constraints within execution of a transaction. Ada does not fare as well in this evaluation. The only timing device that is provided by Ada is a delay. However, Ada 95 provides for periodic processes as well as exception handling for missed timing constraints [BP91].

While two-phase locking was not designed to handle temporal consistency, several real-time techniques have extended two-phase locking to do so [SRSC91, AGM88, HL92]. Each of these techniques assigns priorities to transactions that reflect the timing constraints imposed on them. In [SRSC91] the priority inversion problem is bounded to a single lower priority blocking transaction. Both [AGM88] and [HL92] use priority to resolve conflicts. In general, if a conflict occurs between two transactions, the lower priority transaction yields to the higher priority transaction, aborting if necessary.

Each of the real-time optimistic concurrency control techniques [HCL90a, LS93] was evaluated and compared with other pessimistic techniques based on the number of deadlines

that were missed. Both performed well in this regard compared to the chosen pessimistic techniques. One difference is that [HCL90a] uses transaction priority to actively maintain transaction temporal consistency and [LS93] does not use transaction priority, but rather relies on the increased concurrency to allow more transactions to make their deadlines.

Of the semantic concurrency control techniques, only [WDL93] and [KM93] explicitly address transaction temporal consistency. The RTC language described in [WDL93] allows a programmer to express a full range of timing constraints on transactions. In the SSP protocol described in [KM93], transaction temporal consistency is enhanced by the use of the priority stack scheduler. While the ESR based techniques [WYP92, PHK+93] and the technique described in [WA92] do not explicitly address transaction temporal consistency, the use of correctness criteria that are more flexible than serializability supports the maintenance of temporal constraints (see Figure 1.1).

2.5.2 Data Temporal Consistency

None of the mechanisms that we have described actively supports maintenance of temporal consistency of data. This is one of the major efforts in our research.

2.5.3 Transaction Logical Consistency

Logical consistency of transactions is defined by the way a concurrency control mechanism allows a transaction to use a group of objects. Two-phase locking is more flexible than conservative and strict two-phase locking because they generally hold locks longer. The real-time techniques based on two-phase locking [SRSC91, AGM88, HL92] also maintain transaction logical consistency by preserving serializability of transactions.

Of the object-based semantic techniques, [BR92] and [Wei88] do not discuss logical constraints on transactions. These techniques concentrate solely on the consistency of the data. Type-specific locking is provided in [SS84] to allow a transaction to use more than one data object. The semantics of the application are examined to determine how long a transaction should hold a lock on an object. In [WDL93] the RTC language provides mechanisms for processes to exclusively access objects for as long as is necessary in the particular situation.

The database designer defines transaction logical consistency in the transaction-based semantic concurrency control mechanisms that we have described [GM83, Lyn83, FO89].

The designer defines correctness in an ad hoc manner based on the semantics of each individual transaction. In [ABAK94] the above transaction-based techniques are generalized and correctness is formalized in the notion of relative serializability. In [KM93, WYP92, PHK+93, WA92] transaction logical consistency is also defined by the designer. But in these techniques, the correctness criteria are more structured. The transaction interleavings that are allowed are restricted to maintain limits on imprecision that may accumulate.

2.5.4 Data Logical Consistency

All of the mutual exclusion and serializability techniques maintain data logical consistency. Each of the semantic concurrency control techniques described uses semantics to some extent to maintain logical consistency of objects. The transaction-based techniques define semantic consistency of the data objects that must be maintained. In [BR88], [BR92] and [Wei88] semantics-based commutativity or recoverability is used to maintain serializability. User-defined logical consistency is maintained in [SS84], [WDL93] and [WA92] through a semantic compatibility relation and in [KM93] through the specification of similarity.

Some epsilon-serializability based techniques require that data appear as if it has been accessed serializably [WYP92, PHK+93]. While other techniques that use ESR as a correctness criterion allow non-serializable updates as long as the data will eventually be restored to a consistent state [DP93].

2.5.5 Bounding Imprecision

Imprecision may result from any of the semantic concurrency control techniques that allow non-serializable access to the data. The transaction based techniques [GM83, Lyn83, FO89, ABAK94] handle imprecision in an ad hoc manner. Imprecision is not accumulated or bounded. The same is true for the object-based technique in [SS84]. The correctness criteria defined in [KM92, RP, DP93, WA92] provide the ability to allow non-serializable interleavings of transactions while maintaining a bounded amount of imprecision.

2.5.6 Burden on the User

An important criterion on which to measure semantic concurrency control techniques is the burden that is placed on the user to define the semantics of the application.

All of the transaction-based semantic techniques [GM83, Lyn83, FO89] place a heavy burden on the user because he must know about every transaction in the system. The user must also be able to define compatibilities among all of the transaction types at different points in each transaction. The user is required to view each transaction globally, moreover, the kinds of applications that can utilize this type of concurrency control mechanism are limited.

With epsilon-serializability [RP, DP93], each transaction must specify limits on the amount of imprecision that it can import and export. Similarly, the user specifies allowable amounts of imprecision in the technique described in [WA92]. This is somewhat less difficult than defining transaction compatibilities because the limits defined for each transaction are independent of other transactions. However, in [WA92], it is left up to the user to define the forward resolution set dilating function and the backward resolution set dilating function used to determine compatibility among object operations.

The concept of similarity in [KM92] is defined by the designer for each data item. Thus the transactions need not be known up front, only the data to be used. The object-based techniques, by definition, allow the user to view the application from a more modular perspective. Rather than defining compatibilities among all transactions, the user is required only to define compatibilities among operations on each object. This allows the user to focus on the specific semantics of the object. In [SS84, WDL93] these compatibilities are based solely on the user-defined correctness criteria and therefore the burden is completely on the user to define them. In [BR88, BR92, Wei88] the compatibilities are based on either commutativity or recoverability and are computed dynamically by the concurrency control mechanism. Very little burden is placed on the user in these techniques.

The related work that we have presented provides a rich foundation upon which to build. However, there is no one concurrency control technique that supports logical and temporal consistency of data and transactions as well as the trade-off among them.

Chapter 3

RTSORAC Model

Our semantic locking concurrency control technique is based upon our model of a real-time object-oriented database called RTSORAC. 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 type*, 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.

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 [BOW93]. 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 in-

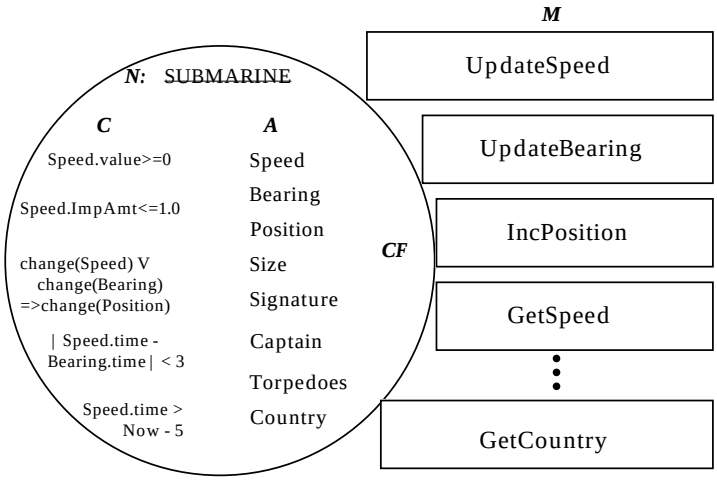

Figure 3.1: Example of **Submarine** Object Type

volved with tracking other ships in a combat scenario. Transactions in this application may also allow for certain amounts of imprecision depending on 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 3.1 illustrates an example of a **Submarine** object type in the database schema.

3.1 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* (see ESR Section 2.4). 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. This time could be computed using techniques described in [PE94]. 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 [PDPW94]. In Figure 3.1 $IncPosition$ 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 3.1 the predicate $Speed.time > Now - 5$ expresses a temporal consistency constraint on the $Speed$ attribute that it should 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 Chapter 4.1.

3.2 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 Chapter 4 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) [BHG86] by requesting and releasing locks in specific orders. Other more flexible transaction locking techniques that do not follow 2PL can also be expressed.

The component C of a transaction is a set of constraints on the transaction. These constraints can be expressed on execution, timing, or imprecision [PDPW94]. 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 [YWLS94]). 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.

For example, assume that 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.

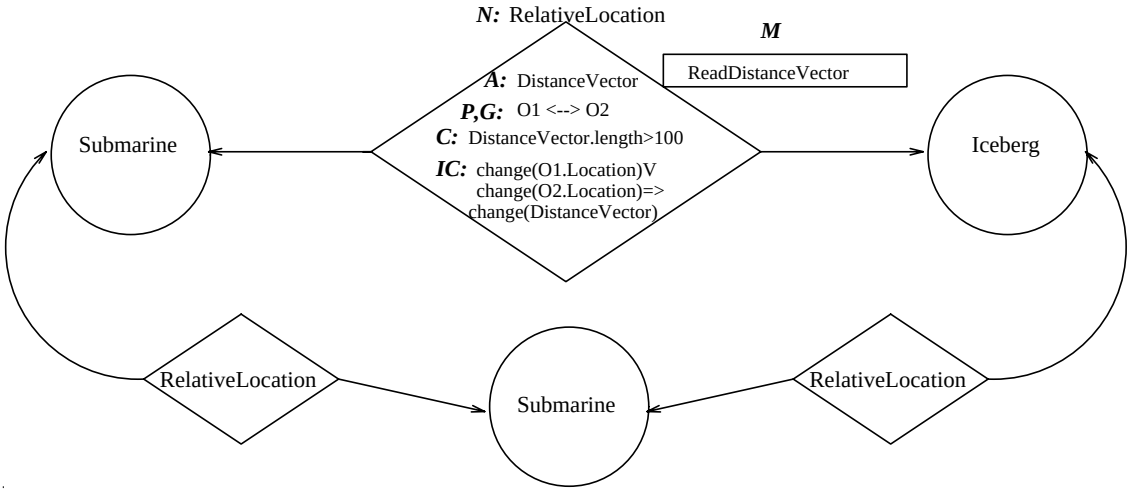

Figure 3.2: Example of **Relative Location** Relationship

3.3 Relationships

Relationships represent aggregations of two or more objects. In the RTSORAC model, a *relationship* consists of $\langle N, A, M, C, CF, P, IC \rangle$. The first five components of a relationship are identical to the same components in the definition of an object. P is the set of participating objects in the relationship. Each participant specifies the type of the participating object. IC is a set of interobject constraints placed on objects in the participant set, and is of the form $\langle PartSet, Pred, ER \rangle$. $Pred$, and ER are as in object constraints, and $PartSet$ is a subset of the relationship's participant set P. The predicate is expressed using objects from the $PartSet$, allowing the constraint to be specified over multiple objects participating in the relationship. Enforcement rules are defined as before, however the operations Op can now include invocations of methods of the objects participating in the relationship.

An example of a relationship that might exist in our submarine application is one called **RelativeLocation** that exists between two **Submarine**s (see Figure 3.2). This relationship contains the attribute *Distance Vector* which indicates the position of the two nautical objects relative to each other in a standard fixed coordinate frame. It also contains interobject constraints on the relative positions of the nautical objects.

Chapter 4

The Semantic Locking Technique

This chapter describes our real-time concurrency control technique for database objects under the RTSORAC model [DW93]. 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.

4.1 Compatibility Function

The compatibility function (CF) component of an object (Chapter 3.1) 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 [BR88], 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 ($RAS(m)$) and a write affected set ($WAS(m)$). 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 ($data\_\epsilon_a$ [DP93]) 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 [SS84].

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 4.1 uses the submarine example of Chapter 3.1 to demonstrate several ways in which the compatibility function can semantically express conditional compatibility of method locks. Example A of Figure 4.1 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

	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:	$CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2)) =$ $(\lvert S_1.value - S_2.value\rvert < (data\_e_{Speed} -$ $Speed.ImpAmt))$	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 4.1: Compatibility Function Examples

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 4.1 illustrates how an attribute can become imprecise. Two invocations of *UpdateSpeed* may occur concurrently if a sensor writes one value and a human user 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 $\lvert S_1.value - S_2.value\rvert$ if this interleaving were allowed.

Example C of Figure 4.1 represents the compatibility function for 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

Semantic Lock Request for m_{req}	**Step**
granted := TRUE	
for every (($m_{act} \in ActiveLocks$) OR	**LA**
((m_{act} in prio queue) AND	
($m_{act}.prio > m_{req}.prio$)))	
if $CF(m_{act}, m_{req})$ then	**LA**$_1$
save m_{act} return arg ImpAmts	
Increment imprecision	**LA**$_2$
else	
granted := FALSE	
endif	
end for	
if not granted then	
Enqueue(m_{req}) in prio queue	**LB**
else	
Add m_{req} to $ActiveLocks$	**LC**
endif	
endif	

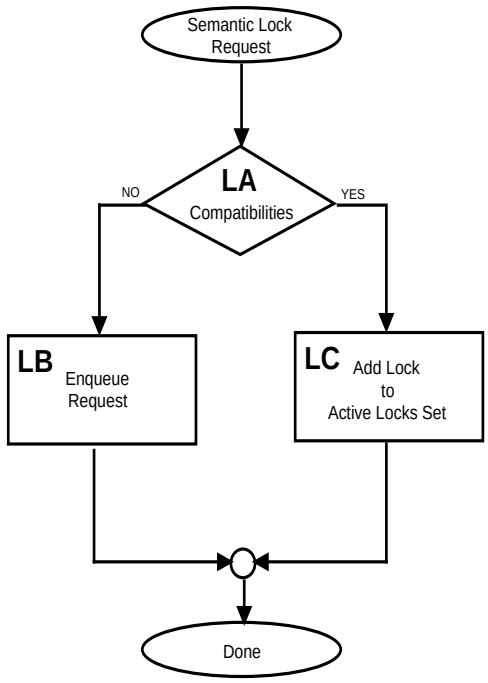

Figure 4.2: Mechanism for Semantic Lock Request

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.

4.2 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 Chapter 3.2, 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 4.2 and 4.3 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.

Method Invocation Req: m_{req}	**Step**
InitialImprecision(m_{req})	**A**
if any Precondition fails then	**B**
Enqueue(m_{req}) in prio queue	**L**
else	
for every $a \in WAS(m_{req})$	$\mathbf{C}_1$
save original $a.ImpAmt$	
$a.ImpAmt := m_{req}.WriteImp(a)$	
end for	
for every $r \in ReturnArgs(m_{req})$	$\mathbf{C}_2$
save original $r.ImpAmt$	
$r.ImpAmt := m_{req}.ReadImp(r)$	
end for	
if already locked then	**D**
Allow m_{req} to Execute	**I**
Semantic Lock Update	**J**
Check the queue	**K**
else	
Semantic Lock Request	**E**
if lock granted then	**F**
Allow m_{req} to Execute	**H**
else	
for every $a \in WAS(m_{req})$	**G**
restore original $a.ImpAmt$	
for every $r \in ReturnArgs(m_{req})$	
restore original $r.ImpAmt$	
for every saved return arg r	
of an active method invocation	
restore original $r.ImpAmt$	
endif	
endif	

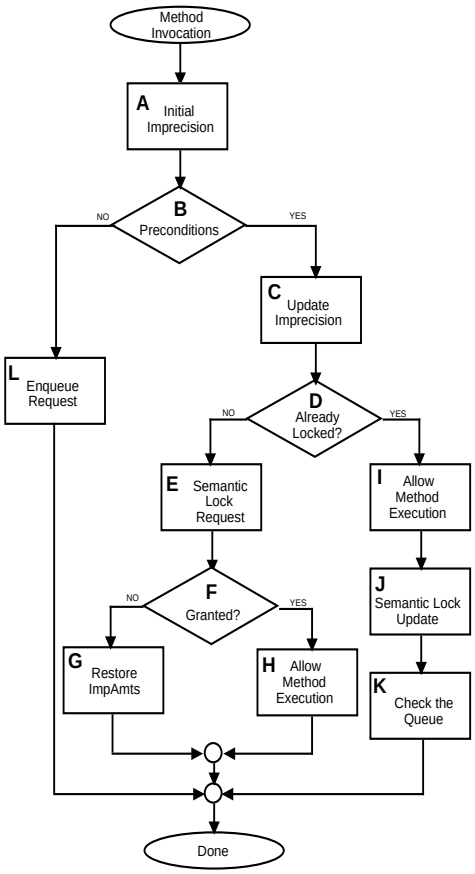

Figure 4.3: Mechanism for Method Invocation Request

4.2.1 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 active locks and with all queued lock requests for method invocations that have higher priority than m_{req} (Figure 4.2, 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).

4.2.2 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 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 4.3 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 4.3, 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} ($m_{req}.WriteImp(a)$). The procedure also computes the amount of imprecision that m_{req} will return through each of its return arguments r ($m_{req}.ReadImp(r)$). 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

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

$$\forall_{a \in WAS(m_{req})}(m_{req}.WriteImp(a) \leq data\_\epsilon_a) \quad (b)$$

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

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 $m_{req}.WriteImp(a)$. Similarly, it updates the imprecision amounts for every return argument r of m_{req} with the value $m_{req}.ReadImp(r)$

(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* [DP93]. 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 Chapter 4.2.1. 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).

4.2.3 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.

Chapter 5

Bounding Imprecision

In this chapter 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) [RP] correctness criteria, defined for object-oriented databases, is met. First we extend the definition of ESR 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 how the restricted semantic locking technique bounds imprecision, using the submarine tracking example.

5.1 Object-Oriented ESR

Recall the ESR definitions of data and transaction safety from Chapter 2.4:

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

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

These definitions of are general; we now define safety more specifically for the RTSORAC real-time object-oriented data 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 (defined in Chapter 2).

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

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

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

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:

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

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

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

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

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

5.2 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 by the initial imprecision procedure before the compatibility function is evaluated (see Chapter 4.2.2).

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

Restrictions

R1: If $a \in WAS(m_1) \bigcap WAS(m_2)$ 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 $a \in RAS(m_1) \bigcap WAS(m_2)$ 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 by 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 4.1B of Chapter 4.1. 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 $|S_1 - S_2| \leq (data\_\epsilon - spec_{Speed} - Speed.ImpAmt)$.

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 in R2b 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 ($m_1.ReadImp(r)$) 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 4.1A of Chapter 4.1 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 are 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. 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.*

5.3 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 4.3) 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 $a \in WAS(m)$. Since the preconditions hold, Step C_1 ensures $a.ImpAmt = m.WriteImp(a)$, and from Precondition (b): $m.WriteImp(a) \leq data\_\epsilon_a$. 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 $a \in WAS(m_1) \bigcap WAS(m_2)$. 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. □

Lemma 2 *If the restricted semantic locking technique is used, then OT-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 $r.ImpAmt$=$m.ReadImp(r)$, and from Precondition (c): $m.ReadImp(r) \leq import\_limit_r$. 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

$RAS(m) \bigcap WAS(m_1) \neq \emptyset$. 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 $RAS(m) \bigcap WAS(m_1) \neq \emptyset$. 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 [DP93], if transactions execute simple operations, the export limit can be omitted and the transaction can rely completely on *data_ε-spec*s for imprecision control. The simple model of transactions of Chapter 3.2 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.

5.4 Example

We use an example of a **Submarine** object, which is an instance of the object type in Figure 3.1 of Chapter 3 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 3.1, 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 4.3. 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 $UpdateSpeed.WriteImp(Speed)$. Because the invocation $UpdateSpeed(10.6)$ writes 10.6 to *Speed* with 0.3 units of imprecision, the initial imprecision procedure computes $UpdateSpeed.WriteImp(Speed) = 0.3$.

The preconditions for the requested $UpdateSpeed(10.6)$ method invocation are tested next (Step B). Precondition (a) trivially holds because $RAS(UpdateSpeed)=\emptyset$. Precondition (b) also holds since $UpdateSpeed.WriteImp(Speed) = 0.3 \leq 1.0$. 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 $UpdateSpeed.WriteImp(Speed)$, 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 4.1 and the example in Chapter 5.2 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 4.1: $CF(UpdateSpeed_1(S_1), UpdateSpeed_2(S_2)) \Rightarrow Speed.ImpAmt := Speed.ImpAmt + |S_1.value - S_2.value|$. 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 executes $UpdateSpeed(10.6)$ (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.

Chapter 6

Implementation

We have implemented the RTSORAC model in a prototype system, developed by John K. Black, that will extend the Open Object Oriented Database System (Open OODB) [WBT92] to support real-time requirements [WPD+]. The prototype executes 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. Before accessing an object, a transaction executes the semantic locking mechanism to provide concurrency control.

6.1 Object Type Implementation

A schema in our prototype is specified by C++ classes for RTSORAC object types. The current implementation provides attributes having only floating point value fields. The compatibility function of an object is implemented as an NxN array where N is the number of methods in the object. The elements of the array indicate compatibility between the methods, where the rows represent requested methods and the columns represent currently active methods. The elements contain either 0 (the methods are never compatible), 1 (the methods are always compatible), or a pointer to a function that uses available information to determine compatibility between the methods.

The implementation provides certain "meta members" in the C++ class representing the object type. These meta members include a wait queue, the compatibility function array, POSIX mutual exclusion locks (mutexes) and condition variables, and member functions

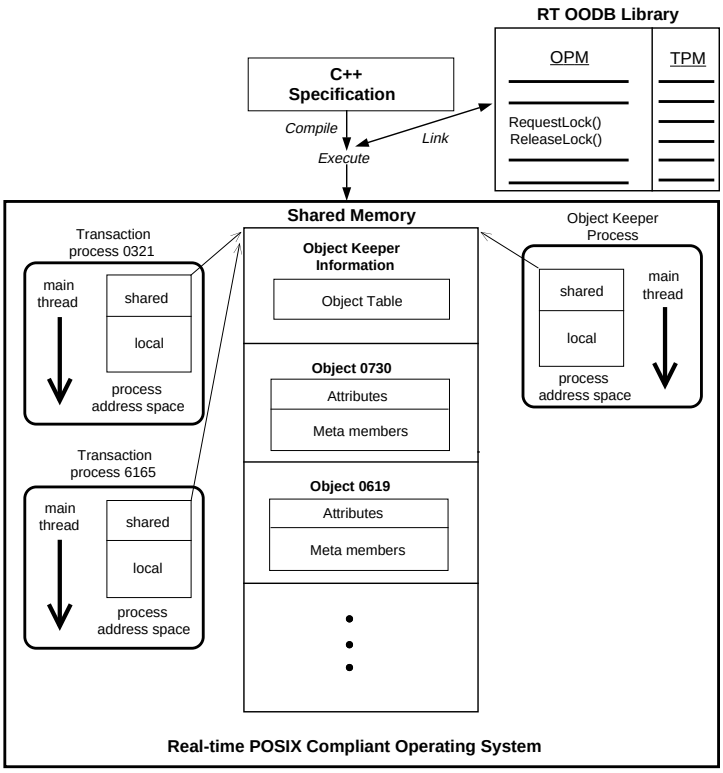

Figure 6.1: Object Management Implementation

to request and release locks on the object (*RequestLock* and *ReleaseLock*). These meta members are provided to assure that only one transaction may execute the semantic locking mechanism at a time.

6.2 Transaction Implementation

Transactions in the prototype system are C++ programs that include the schema file of object type declarations. Each transaction program is compiled into a POSIX process (or a thread within a process). Each process maps all database objects, which reside in shared memory, into its own address space. The process uses calls to an object's meta member *RequestLock* to lock an object while using it. Once the object is locked, the transaction calls the object's methods as if the object were in the transaction's own address space. A transaction process uses calls to the underlying operating system to set its priority and to set alarms for start times and deadlines.

6.3 Shared Memory Management

In the prototype system, an *object keeper* process creates a shared main memory segment at system startup. This keeper process may load the shared segment with object instances, either by restoring previously archived objects, or by instantiating new objects. Transaction processes use the POSIX shared memory capabilities to map the shared segment into their own virtual address spaces (see Figure 6.1), thereby gaining direct access to object instances. Transactions use an overloaded C++ `new` operator to dynamically place objects in the shared segment or to locate existing objects by name. To do this, part of the shared segment is reserved at a well-known offset for use by the system as an *object table*. The table associates each object's name with the object's offset from the shared segment's base address. The table also stores object type information. The special `new` operator automatically manages the object table and uses it to translate object names to offsets. From this offset, the `new` operator creates a properly typed pointer to the object in the shared memory segment and returns this pointer to the transaction. There is also an overloaded `delete` operator for removing objects.

6.4 Semantic Locking Mechanism Implementation

The implementation of the semantic locking mechanism is based on the pseudocode of Figure 4.3. The implementation currently allows only simultaneous method invocations, and not lock requests for future method invocations (see Chapter 4). A transaction requests a semantic lock for a method invocation by calling the meta member function *RequestLock*, specifying the method and the arguments for the requested invocation. The meta member function acquires the POSIX mutex for access to the object's meta data. When the mutex is granted, the *RequestLock* function attempts to acquire a semantic lock for the transaction (See Figure 4.3). If the lock is granted, the request is placed in the active locks set and the transaction can immediately continue execution and call the method. If the lock is not granted, the transaction is placed in the object's wait queue and is suspended. The suspended transaction will be awakened to retry its lock request when another lock is released. Whether or not the lock is granted, the transaction releases the mutex at the end of the *RequestLock* function. Note that mutexes are used to ensure mutual exclusion only for each object's meta members during the semantic locking mechanism execution.

Transaction access to object attributes is controlled with semantic locks.

A transaction explicitly releases the locks that it holds by calling the *ReleaseLock* meta member function on the object. This meta member function removes the method invocation from the object's active locks set. It then broadcasts on a POSIX condition variable to awaken all of the suspended transactions in the object's wait queue so they may retry their locks requests. The use of a real-time scheduler provided by the operating system assures that the awakened transactions make their lock requests in priority order.

Chapter 7

Evaluation

We utilized the prototype system described in Chapter 6 to conduct performance tests in which we compared two versions of our semantic locking mechanism with other object based concurrency control techniques (exclusive locking, read/write locking and commutative locking). Each test involved generating a set of synthetic system configurations and a set of synthetic workloads. On each system configuration, we executed the corresponding workload using each of the concurrency control mechanisms. The results of these tests indicate, in general, that our semantic locking technique, in both forms, maintains transaction temporal consistency better than the other concurrency control techniques. The results also point out under what semantic conditions our semantic locking technique best maintains data temporal consistency. In this chapter we first describe the construction of our testbed. We present the performance model and performance parameters and compare them to a well-known performance model to show the validity of our testing. We go on to explain why we chose to compare our semantic locking technique with the techniques listed above. We describe the measurements that we used to compare the different concurrency control techniques, and briefly describe how the testing was performed. Finally, we present the results of the tests and analyze them.

7.1 Testbed Construction

Figure 7.1 illustrates how a system configuration and a workload is generated. The range file stores the data ranges from which the parameters are randomly generated. The workload and configuration generation program reads from the data ranges file and uses a seed value

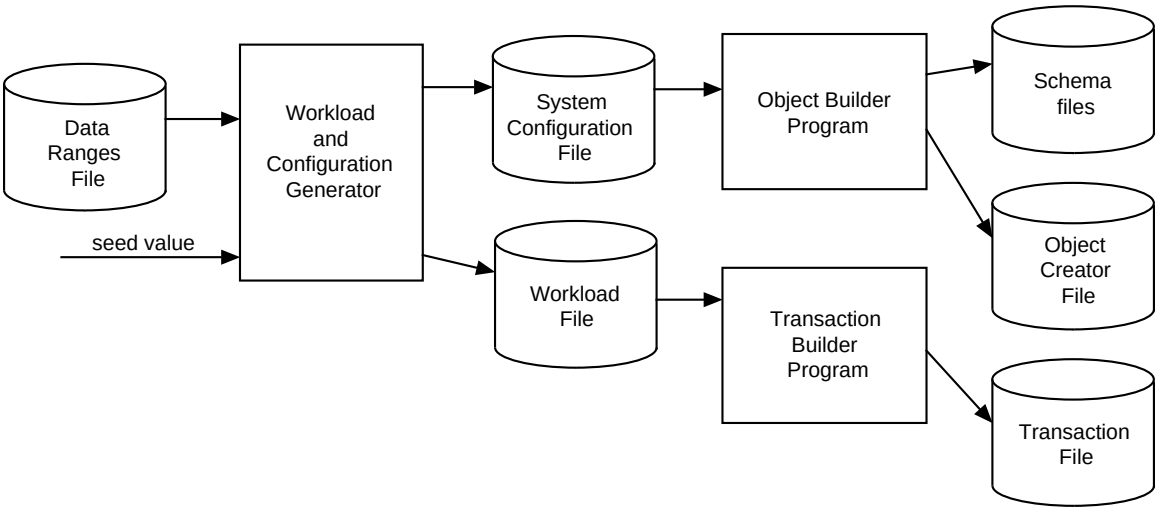

Figure 7.1: Construction of Testbed Configuration

to produce a random number within the specified range for each parameter (see Section 7.3 for performance parameters). The workload and configuration program produces the object parameters and the transaction parameters in the system configuration file and the workload file respectively. The object builder program then reads from the system configuration file and produces schema files that contain the C^{++} specifications of the objects in the system. The object builder program also produces a file containing information for storing the objects in shared memory (object creator file). The transaction builder program reads from the workload file and produces a file containing C^{++} code for the transactions of the workload specification.

Once the system configuration and the workload are generated, the test is run using the prototype system described in Chapter 6. Figure 7.2 illustrates how a test is run. First the shared memory segment is created. Then the object creator program is compiled including the schema files. The object creator program runs placing the objects of the system configuration into the shared memory segment. When the objects are in shared memory, the controller program, which is compiled to include the schema files and transaction code, runs as the controller process. The controller process maps the shared memory segment into its own memory space and spawns threads representing the transactions of the workload. The transaction threads run, accessing objects using the chosen concurrency control mechanism. The controller process reports the results of the test to a statistics file. We repeated this procedure for each test that we performed, changing the seed value to get different random

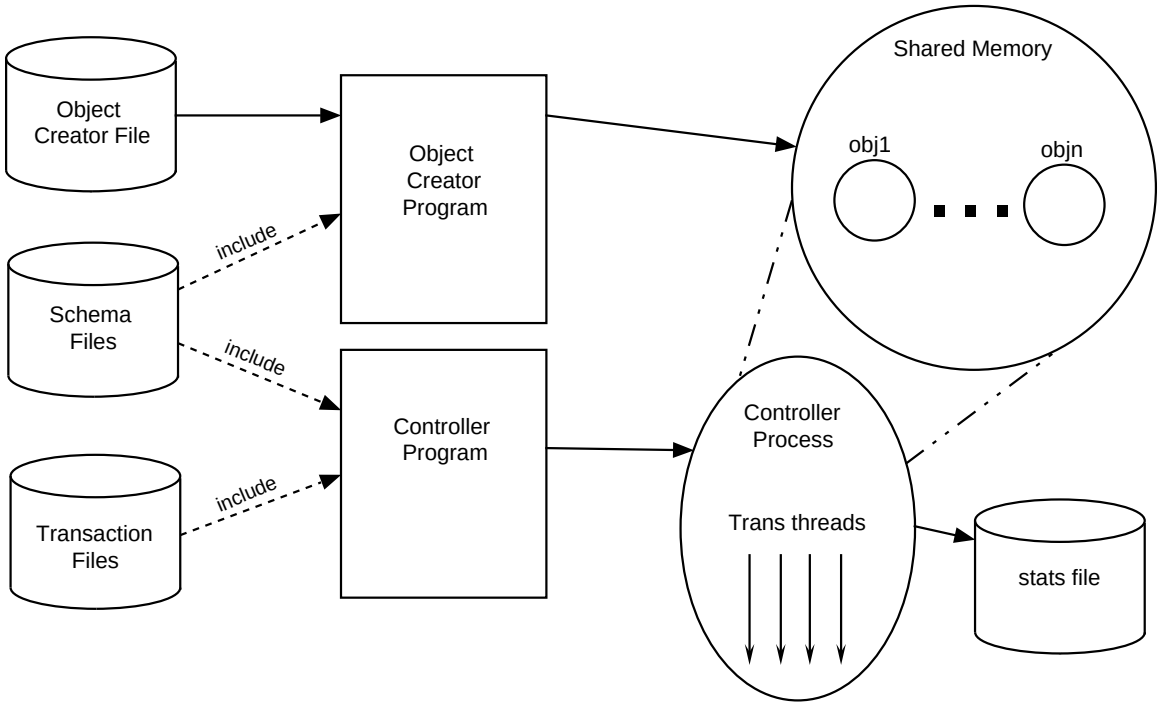

Figure 7.2: Running a Test

configurations, and changing the range file to vary specific parameters.

7.2 Performance Model

As a baseline model for our testing, we used the canonical concurrency control simulation model of [ACL87], with some modifications to accommodate the semantics unique to our model and technique. Figure 7.3 displays the logical queuing model of [ACL87] which is central to their simulation model for concurrency control algorithm performance. We will refer to the simulation model of [ACL87] as the Agrawal model for simplicity. The terminals in Figure 7.3 represent sources of transactions. When a transaction originates at a terminal and the maximum number of transactions are active, the new transaction enters the ready queue. When the transaction comes off the ready queue, it enters the concurrency control queue (cc queue) and makes its concurrency control requests to the concurrency control module. If the request is granted, the transaction goes to the object queue to access the requested objects, cycling through all of the objects in the request. The transaction returns to the cc queue to make its next request. If a request is denied, the transaction enters the blocked queue where it waits to reenter its request.

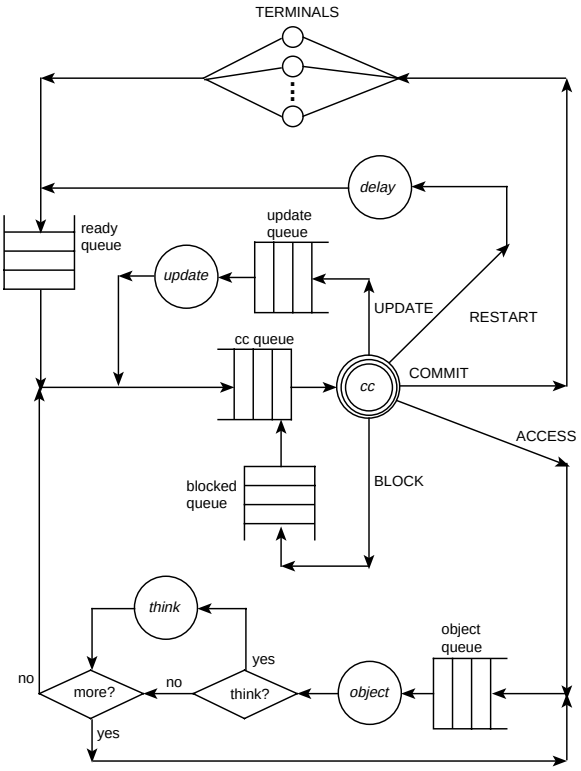

Figure 7.3: Agrawal Performance Model

To test our semantic locking technique, we made several modifications to the Agrawal model to handle the required semantics and the real-time aspects of our technique. Here we describe how our performance model maps to the Agrawal model of Figure 7.3. Our controller program (see Figure 7.2 represents the terminals of the Agrawal model, starting transactions at specified start times. Rather than place a maximum number of transactions (multiprogramming level) on our system, we use the range of start times to represent interarrival time, which is a better measure of system load than multiprogramming level in real-time databases [AGM88, HSTR89]. Thus, in our model, the ready queue is not necessary.

A major difference between our performance model and the Agrawal performance model is that ours is modularized so that each object has its own concurrency control queue. In our model, the mutex that guards the semantic locking mechanism and the meta data of an object (see Chapter 6) represents the cc queue of Figure 7.3. The priority wait queue in our technique represents the blocked queue of Figure 7.3 with the additional semantics of real-time priority. The semantic locking mechanism of our technique represents the concurrency

Parameter	Meaning
db_size	Number of objects
tran_size	Mean size of transactions
max_size	Size of largest transaction
min_size	Size of smallest transaction
write_prob	Probability that transaction writes object
int_think_time	Mean intratransaction think time
restart_delay	Mean transaction restart delay
num_terms	Number of terminals
mpl	Multiprogramming level
ext_think_time	Mean time between transactions
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
num_cpus	Number of CPUs
num_disks	Number of disks

Table 7.1: Performance Parameters for Agrawal Performance Model

control module of the Agrawal model. It examines one concurrency control request at a time and when granted, the requested actions are performed immediately on the object. Thus, our performance model has no object queue or cycle for more requests that are seen in Figure 7.3. The results of a concurrency control request in our model are simpler than in the Agrawal model, allowing only the actions BLOCK and ACCESS. Our transactions do not restart and they update in place. A commit of a transaction is assumed in our model after all of its locks are released.

7.3 Performance Parameters

The parameters of our performance model are based on the parameters of the Agrawal model displayed in table 7.2. There are several parameters that do not apply to our system. For instance, we do not have a parameter to represent the number of disks (*num_disks*) or the object I/O time (*obj_io*) because our system is a main memory database. The number of CPUs (*num_cpus*) in our model is one, and the controller program (see Figure 7.2) is the only source of transactions (*num_terms*). Since our transactions do not restart, we have no parameter to represent *restart_delay*. On the other hand, our model requires parameters that do not exist in the Agrawal model because our technique examines object semantics to determine concurrency control. In this section, as we present our performance parameters, we point out if and how they map to the Agrawal model performance parameters.

Objects	
no. attribs	1-5
no. methods	2-5

A

Attributes	
value	1.0-10.0
time	0
imp amt	0.0
imp limit	1.0-10.0
avi	1-10

B

Methods	
affected sets	0-1
exec time	1-3

C

Figure 7.4: System Configuration Tables

System Configuration. The system configurations that were generated in our testing consisted of groups of data objects. Each configuration was made up of ten objects (*db_size* in Table 7.2), each with randomly generated attributes, methods and constraints. The compatibility function for each object was generated based upon the concurrency control protocol used and the semantics of the object (See Section 7.4).

Figure 7.4 illustrates how each of the parameters in the system configuration was generated. In general, the ranges of values for the parameters were chosen so that the system configurations were complex enough to produce interesting results, while remaining reasonable. Chart A shows that for each object, the number of attributes was between 1 and 5, and the number of methods was between 2 and 5. Figure 7.4B shows how the fields of each attribute were generated. The value field was generated randomly from the range of numbers shown. The time field of the attribute was set at the time the test started. In the chart, the zero represents the time relative to the time the system configuration was built. The amount of imprecision initially in the attribute (at the start of the test) was zero. Two constraints relating to the attributes were also generated randomly. The imprecision limit for the attribute was between 1.0 and 10.0. The *avi* (absolute validity interval) was generated from a range of relative times (1 to 10 seconds) and was then added to the current absolute time when the system configuration was built. There are no analogous Agrawal parameters (Table 7.2) to those in Figure 7.4B because the parameters in Figure 7.4B represent specific semantics of the data.

Figure 7.4C displays parameters for each method of an object. The affected sets (read affected set and write affected set) were generated randomly so that for each attribute in the object, a value of either 0 or 1 was randomly chosen to specify if the attribute was

Transactions	
no. meth invocs	1-5
start time	4-35
deadline	12-25
exec time	computed
slack time	computed
priority	computed

A

Method Invocations	
object	sys config
method	sys config
temporal	0-1

B

Method Invocation Arguments	
imp limit	1.0-10.0
imp amt	0.0
value	1.0-10.0
time	0

C

Figure 7.5: Workload Tables

in the affected set. The affected set parameter is analogous to the *write_prob* parameter of the Agrawal model. Because the methods of our RTSORAC model permit more than simple reads and writes, write probability was not sufficient for our testing environment. The execution time for each method was generated as an integer number of KiloWhetstones [DSW90]. [1] This execution time is analogous to the *int_think_time* parameter of Table 7.2. It can also be considered to be analogous to the *obj_cpu* parameter because each access of an object in our system is a method invocation.

To facilitate definition of object semantics in our testing environment, we made a simplifying assumption regarding the methods of an object:

Assumption 1 *For every attribute a in the read affected set of a method m, there is a return argument $r_{m,a}$ that returns the value read by m, and for every attribute a in the write affected set of the method m, there is an input argument $i_{m,a}$ that stores a value to be written by m. We also assumed that the only execution performed by a method was done by the reads and writes associated with its arguments.*

The arguments of a method and their types were determined by the randomly generated affected sets. For example, if an attribute a was in the read affected set of a method m, then m had an return argument r that returned the value of a.

Workload. Generation of a workload for our performance tests involved building transactions. Because the transactions had to access data by invoking specific methods, the

[1] This execution time was later converted to seconds and nanoseconds based on testing on the prototype implementation.

workload had dependencies on the system configuration used. Therefore, each system configuration had a corresponding workload configuration. Each test that we performed involved 20 transactions accessing a single system configuration. Figure 7.5 displays the parameters that were used to build transactions for the workload. Chart A in the figure indicates that for each transaction the number of method invocations was generated randomly from a range of 1 to 5. We used this parameter to represent transaction size (*tran_size*, *max_size* and *min_size* of Table 7.2). For the start time and deadline of each transaction, random relative times (in seconds) were generated from the ranges indicated in the chart. They were relative to some initial starting time for the entire test. The start time determined the *ext_think_time* of the Agrawal model, and we added the deadline parameter to account for real-time semantics. We used the range of start times to represent system load, so we did not use the *mpl* parameter of Table 7.2. The execution time of a transaction was calculated by adding the execution times of each of the methods that the transaction invoked. The slack time was calculated by subtracting the execution time from the relative deadline. The priority of the transaction was determined based on a least slack time priority assignment scheme, which has been shown to be optimal under certain conditions [CSK88].

Figure 7.5B shows the parameters for each method invocation of a transaction. It is for these parameters that the workload generation required knowledge of the system configuration. Because the Agrawal model involved only reads and writes, our method invocation parameters were not based on any Agrawal parameters. For the generation of each method invocation, first an object was chosen randomly from among all of the objects in the system configuration. Then a method was chosen randomly from among all of the methods of the chosen object. For each argument to the chosen method (See Figure 7.5C), if it was a return argument, an imprecision limit was generated randomly from a range of 1.0 to 10.0. If the argument was an input argument, a value was generated from a range of 1.0 to 10.0. The time field for the input argument was the time at which the write actually took place and the initial imprecision amount for the input argument was zero. The generation of the method invocation also randomly determined (from 0 or 1) whether or not the transaction required temporally consistent data to be returned by the invocation.

Each transaction in a given workload requested locks using a two-phase locking scheme. The transaction requested a lock when it was needed (just before invoking the method) and the transaction held the lock until the end of its execution. Transactions that missed their

Exclusive Locking	$CF(m_1, m_2) =$	$FALSE$
Read/Write Locking	$CF(m_1, m_2) =$	$(WAS(m_1) = \emptyset) AND (WAS(m_2) = \emptyset)$
Commutativity Locking	$CF(m_1, m_2) =$	$(WAS(m_2) \cap (WAS(m_1) \cup RAS(m_1)) = \emptyset)$ $AND ((RAS(m_2) \cap WAS(m_1)) = \emptyset)$
m_1=requested method, m_2=active method $RAS(m)$=read affected set of m, $WAS(m)$=write affected set of m		

Table 7.2: Compatibility Function for Comparison Techniques

deadlines were aborted and not restarted.

7.4 Comparison Techniques

One unique feature of our semantic locking technique is the way in which the technique defines conflict between transactions. Our user-defined compatibility function defines conflict between methods based on object semantics and system characteristics. To demonstrate the performance of our technique, we have chosen to focus our testing on how conflicts are defined. That is, we have compared our technique with other concurrency control techniques that define conflict in various ways.

We have implemented each of the object locking techniques by defining the compatibility function accordingly. Table 7.2 shows the compatibility function for each of the techniques with which we compared our semantic locking technique. Exclusive locking defines conflict by mutual exclusion. Only one transaction may access an object at a time. The corresponding compatibility function in Table 7.2 is therefore always false; allowing no methods within an object to interleave. Read/write locking of objects allows multiple readers of an object, but only one writer at a time. The compatibility function for read/write locking ensures that two methods that have common attributes in their write affected sets do not interleave. Commutativity of methods, as defined in [BR88], allows methods to interleave only if the intersections of the affected sets of the methods involved are empty. The compatibility function for commutativity locking in Table 7.2 ensures that if the requested method writes an attribute, no active method reads or writes the attribute, and if the requested method reads an attribute, no active method writes the attribute.

In general, the compatibility function for our semantic locking mechanism is defined by the designer of the object, based on the semantics of the specific application. To demonstrate how our technique can express the trade-off between temporal and logical consistency, we

implemented two versions: one in which logical consistency was chosen over temporal, and the other in which temporal consistency of data was chosen over logical. For ease of description, we refer to the former as the *semantic-logical* technique and we refer to the latter as the *semantic-temporal technique.*

To implement the semantic-logical technique, we used the semantics of the compatibility function restrictions of Chapter 5.2 for our generated objects. Thus, logical consistency was defined by the OESR correctness criterion. To implement the semantic-temporal technique, we used the same semantics as the semantic-logical technique, except that the semantic-temporal technique allowed concurrency that could violate OESR in order to preserve the temporal consistency of the data. For example, if in the submarine example of Chapter 3, a currently active *GetSpeed* method was reading the temporally invalid *Speed* attribute, the semantic-temporal technique would have allowed a requested *UpdateSpeed* method to write to the *Speed* attribute even if it violated the imprecision limits of the transaction that requested the *GetSpeed* method.

For the semantic-logical technique, given a currently active method m_1 and a requested method m_2, the compatibility function $CF(m_1, m_2)$ started out as $TRUE$ and the object builder program of Figure 7.1 iteratively added clauses as follows:

a: $\forall_{a \in (WAS(m_1) \cap WAS(m_2))} CF := CF \ AND \ |i_{m_1,a}.value - i_{m_2,a}.value| \leq data\_\epsilon_a - a.ImpAmt$

b: $\forall_{a \in (WAS(m_1) \cap RAS(m_2))} CF := CF \ AND \ |a.value - i_{m_1,a}.value| \leq import\_limit_{r_{m_2,a}} - r_{m_2,a}.ImpAmt$

c: $\forall_{a \in (RAS(m_1) \cap WAS(m_2))} CF := CF \ AND \ |a.value - i_{m_2,a}.value| \leq import\_limit_{r_{m_1,a}} - (r_{m_1,a}.ImpAmt + i_{m_2,a}.ImpAmt)$

Notice that these three cases are directly analogous to the compatibility function restrictions of Chapter 5. Recall Assumption 1 which involves the direct relationship between arguments of a method and the attributes of its object. In case **a** above, $i_{m_1,a}.value$ represents the value written to attribute a by method m_1 and $i_{m_2,a}.value$ represents the value written to a by m_2. The difference between these values is the potential imprecision that will be introduced if methods m_1 and m_2 interleave. In the read/write conflict in cases **b** and **c**, $a.value$ is the value of the attribute a before the write takes place and $i_{m,a}.value$ is the value of a after the write takes place. The difference between these values represents

the potential imprecision that will be returned through the return argument $r_{m,a}$ if the methods are allowed to interleave.

For the semantic-temporal technique, the object builder program of Figure 7.1 built the compatibility function the same as above except in case **c**. If, in the course of building the compatibility function, case **c** arose, then the object builder program added the following to the compatibility function:

$$OR((Now - a.time) > a.avi)$$

Adding this clause allowed the requested method to acquire a semantic lock and update the temporally inconsistent data even if it violated the logical consistency of the reading method.

Each of the five object locking concurrency control techniques was implemented in our prototype system described in Chapter 6. For the implementation of exclusive locking, read/write locking and commutativity locking we used a simplified version of the semantic locking technique in which all of the steps in the technique that involved testing or accumulation of imprecision (Steps A, C and G of Figure 4.3) were left out because none of these techniques allows any imprecision. This removed any unnecessary overhead from the comparison techniques so that they were better represented. The temporal precondition was left in the comparison techniques so that any difference that was found among the techniques could be attributed to the way in which conflict was defined, and not to differences in how locks were acquired.

7.5 Performance Measurements

Traditionally the measure of a concurrency control protocol is the throughput of transactions [ACL87]. However, because our technique was designed for real-time applications, it is more important to measure temporal consistency than it is to measure throughput. One way to measure temporal consistency of transactions in a real-time database is through the percentage of transactions that miss their deadlines (deadline miss ratio) [HSTR89, AGM88]. To measure the temporal consistency of the data we calculated the percentage of method requests that returned temporally invalid data to its transaction (temporal inconsistency ratio) [Son92].

Test Suite	Deadline Miss Ratio	Temporal Inconsistency Ratio
	DL1: Method Invocations	TI1: Baseline
	DL2: Method Execution	TI2: Method Execution
	DL3: Deadline	TI3: Absolute Validity Interval
	DL4: Allowable Imprecision	TI4: Allowable Imprecision

Table 7.3: Tests Performed

7.6 Testing

For each test that we performed we generated 15 system configurations and 15 corresponding transaction sets. The results of each test were averaged over these 15 trials producing a 95% confidence level with an error of at most 5% for deadline miss ratio and less than 1% for temporal inconsistency ratio (unless otherwise specified). We executed a test for each of the five concurrency control protocols that we compared. We also varied the interarrival time of transactions to illustrate how the techniques perform under varying system loads. We used the range of start times for a transaction as a measure of interarrival time. That is, the smaller the range of start times for a set of transactions, the closer the interarrival time and therefore the heavier the load. Table 7.3 summarizes the tests that we performed.

7.6.1 Deadline Miss Ratio

We performed four test suites to measure deadline miss ratio, each to highlight a particular parameter of the testing.

Test Suite DL1: Method Invocations. The first test was chosen to illustrate how the length of transactions affects concurrency control. We used the number of method invocations in a transaction to represent transaction length. A short transaction had a randomly generated number of method invocations from 1 to 3. A medium length transaction had from 4 to 6 method invocations. A long transaction had from 7 to 9 method invocations.

Test Suite DL2: Method Execution. The length of the methods invoked by a transaction is another way of examining the effect of length of transaction. We varied the execution time of methods so that it was randomly chosen from a range of 1 to 3 KiloWhetstones for

short methods, 5 to 8 KiloWhetstones for medium length methods, and 10 to 15 KiloWhetstones for long methods.

Test Suite DL3: Deadline. We varied the length of the transactions' deadlines in order to examine how the concurrency control mechanism reacts to different real-time environments. The deadlines for transactions were randomly chosen from a range of 8 to 11 seconds for short deadlines, 12 to 15 seconds for medium deadlines, and 17 to 20 seconds for long deadlines.

Test Suite DL4: Allowable Imprecision. We wanted to examine how our own semantic locking mechanism works in applications with varying amounts of allowable imprecision. For this test suite, we did not run the other concurrency control techniques (exclusive, read/write, and commutativity) because none of them allows any imprecision. We varied the imprecision limits for attributes and return arguments. The first test allowed no imprecision. Then the limit for "medium" imprecision was randomly chosen from a range of 1.0 to 5.0, and the limit for "high" imprecision was randomly chosen from a range of 6.0 to 10.0.

7.6.2 Temporal Inconsistency Ratio

We measured temporal inconsistency by examining the percentage of all method requests that read temporally inconsistent data. In order to do this, we had to change the system so that transactions did not abort when they missed their deadlines, but rather continued until complete. We found that if transactions were allowed to abort, a concurrency control mechanism that missed a lot of deadlines appeared to preserve temporal consistency better than a mechanism that allowed more deadlines to be made. This was because the aborted transactions stopped at a time when they were most likely to read temporally inconsistent data.

Without transaction aborts, another problem emerged. In the tests for deadline miss ratio, deadlock was avoided because transactions had a maximum amount of time to run, and then they aborted. Without deadlines, there was the possibility that deadlock would occur and the tests could not be run. In order to alleviate this problem, we used the method of requesting locks in a specified order. The objects and their methods in the

system configuration were given a total ordering and the transaction builder program of Figure 7.1 built the transactions so that they requested locks in the specified order.

The total ordering alleviated most of the deadlock situations, but the priority queue in our system produced another source of possible deadlock. Because our semantic locking mechanism required that a requested method be compatible with all active methods and all queued requests of higher priority, it was possible for a transaction $T1$ to be blocked by a method request of higher priority transaction $T2$ that was blocked by an active method of $T1$. In order to prevent these situations from stopping our tests, we placed a very long deadline on all transactions (5 minutes). Those transactions that were not complete after this long deadline were assumed to have been stuck in deadlock. From these transactions, we only counted the method requests that successfully returned data.

We performed four test suites (Table 7.3 TI1-TI4) to measure temporal inconsistency ratio. First we produced the baseline test to compare our semantic-temporal and our semantic-logical techniques with the other object-based concurrency control techniques. We then performed three other test suites to examine how our semantic-logical technique performed under different conditions. We looked at varying method execution time, absolute validity interval and allowable imprecision. We chose to use our semantic-logical technique in these tests as opposed to our semantic-temporal technique because we wanted to use a technique with bounded imprecision.

Test Suite TI1: Baseline Test. In the baseline test we chose execution time, from a range of 5 to 8 KiloWhetstones, and absolute validity interval, from a range of one to three seconds. The rest of the parameters were as set out in Figures 7.4 and 7.5. We performed this test to compare all of the five concurrency control techniques in a typical configuration.

Test Suite TI2: Method Execution. We examined the performance of our semantic-logical technique with low method execution (1 to 3 KiloWhetstones), medium method execution (5 to 8 KiloWhetstones), and high method execution (10 to 15 KiloWhetstones). The purpose of this test was to see if the length of the methods in the system affects the temporal consistency of the data read by transactions in our technique.

Test Suite TI3: Absolute Validity Interval. The absolute validity interval is directly related to data temporal consistency because it is this interval that defines the temporal con-

sistency of the data. Thus, we chose to test how our semantic-logical technique performed with varying values for absolute temporal validity. We first examined the performance when attributes were considered temporally valid for a very short period of time (low *avi*, 0 to 1 second). The medium absolute validity interval test chose *avi* from a range of 1 to 3 seconds. For the long absolute validity interval, the *avi* for attributes was randomly chosen from a range of 3 to 5 seconds.

Test Suite TI4: Allowable Imprecision. The fourth suite of tests examined how temporal consistency in our semantic-logical technique was affected by varying amounts of allowable imprecision. The first test allowed no imprecision. The second test in the test suite allowed a medium amount of imprecision, between 1.0 and 5.0. The high imprecision test allowed imprecision from 6.0 to 10.0.

7.7 Results

The results of our deadline miss ratio performance tests are displayed in Figures 7.6-7.17. In most cases the two applications of our semantic locking mechanism missed fewer deadlines than the other concurrency control mechanisms. Figures 7.18-7.21 display the results of the tests measuring temporal inconsistency ratio.

7.7.1 Deadline Miss Ratio Results

Test Suite DL1: Method Invocations. Taking the 5% error into account, there was virtually no difference among the five concurrency control mechanisms with both very short transactions and very long transactions (See Figures 7.6 and 7.8). There was a large difference between the overall results for short transactions and the overall results for long transactions. When transactions were very short, very few deadlines were missed, and when transactions were long, a very high percentage of deadlines was missed. This overall result was what we would have expected. One reason that we did not see any significant difference among the concurrency control techniques for short and long transactions was that the values we chose to represent long transactions and short transactions were extreme.

The test involving medium length transactions (Figure 7.7) showed some more interesting results. The maximum error for this test was 9%, but even taking that error into

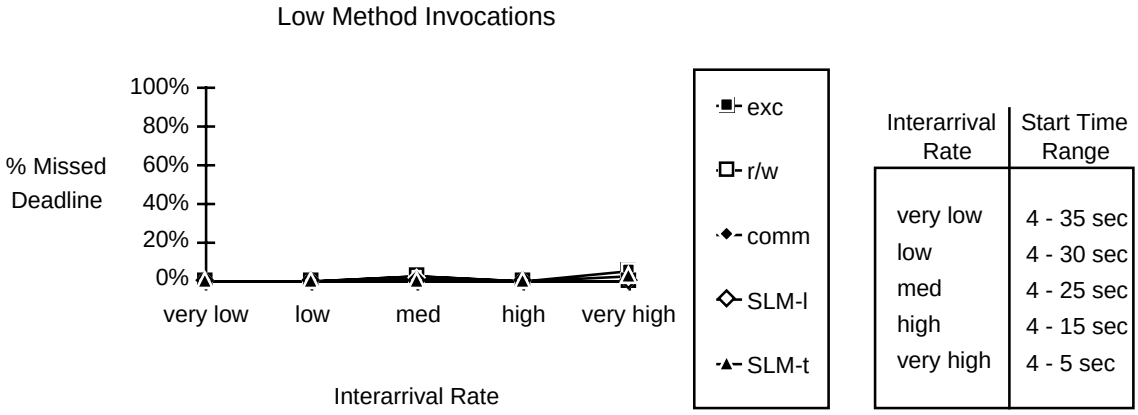

Figure 7.6: Low Method Invocations

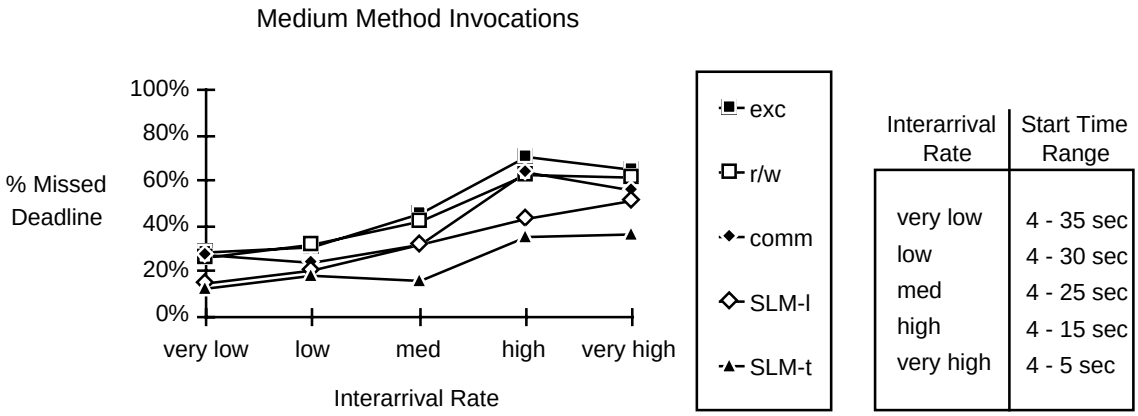

Figure 7.7: Medium Method Invocations

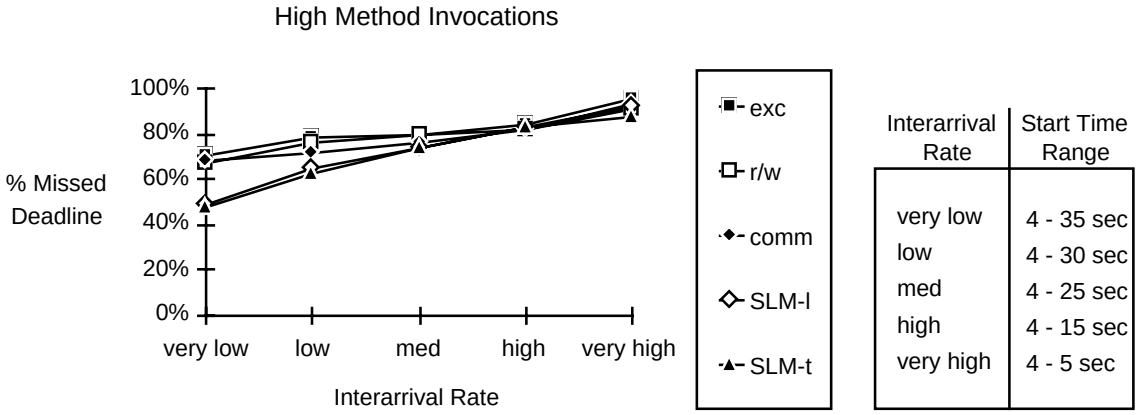

Figure 7.8: High Method Invocations

account, we can see that our semantic techniques performed better than the others. And at high interarrival rates, the semantic-temporal technique missed slightly fewer deadlines than the semantic-logical technique. This result may be due to the fact that with very high contention for the processor (high interarrival rates) there were more chances for methods of different transactions to conflict. It was therefore more likely that the read/write conflict that allowed the semantic-temporal technique to violate logical consistency would occur.

Test Suite DL2: Method Execution. Figures 7.9-7.11 display the results of the tests performed to examine the effect of method length. In general, the results indicate that the longer the methods of a transaction, the more deadlines are missed, and the more alike the concurrency control techniques that we tested. For long methods (Figure 7.11), all five techniques were essentially the same, missing almost all of the transactions. In this test, the interarrival rate of transactions had very little effect.

The test for medium length methods (Figure 7.10) showed a greater variance of performance among the concurrency control mechanisms due to varying interarrival rate. At very low processor contention (low interarrival rate), the semantic techniques performed better than the others. As the interarrival rate increased, the difference among the techniques diminished and all five techniques had a high miss ratio.

Short methods produced an even greater difference among the concurrency control techniques tested. The miss ratio of the two semantic techniques remained very low regardless of the interarrival rate. The performance of the other techniques was very close to the performance of the semantic techniques at low interarrival rates. As the interarrival rate increased, they had higher miss ratios than the semantic techniques.

One possible explanation for these results was that as method execution time of the methods increases, the possibility of the real-time scheduler finding a feasible schedule, one that met all of its deadlines, decreased. Thus, at high method execution (Figure 7.11), there was very little difference among the five concurrency control techniques because there were very few feasible schedules. As the methods became shorter (Figures 7.10 and 7.9), there was more difference among the concurrency control techniques' performance. Recall Figure 1.1) which we used to illustrate how semantic concurrency control techniques can meet more deadlines because they provide flexibility in finding feasible schedules. The results of Figures 7.9-7.11 accentuate that point: when there were more feasible schedules (shorter

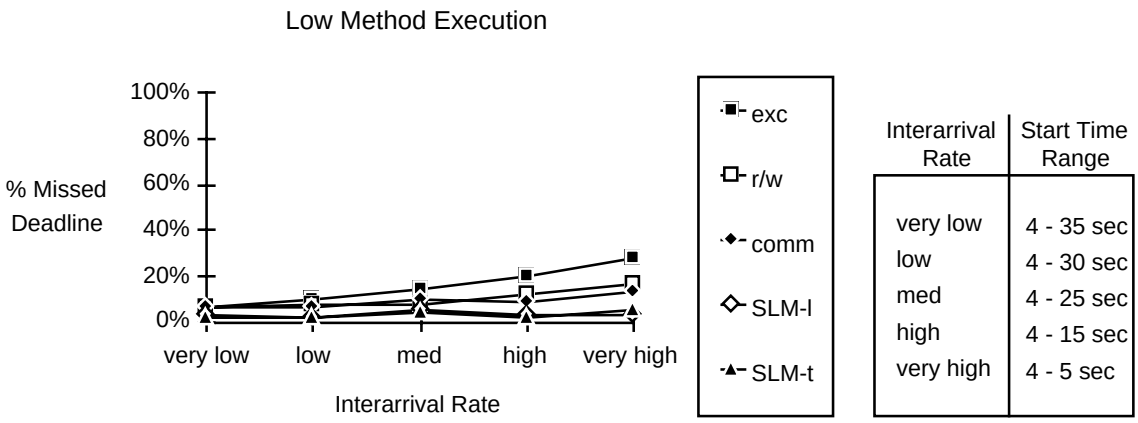

Figure 7.9: Short Methods

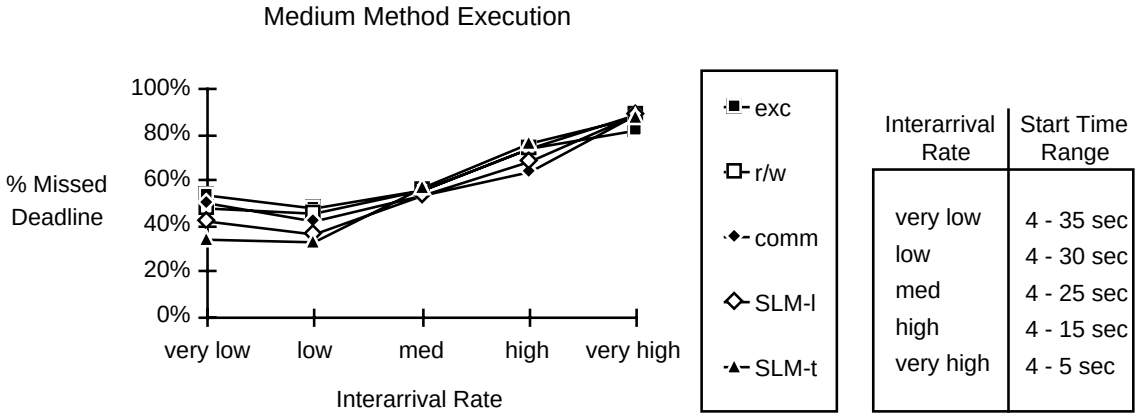

Figure 7.10: Medium Length Methods

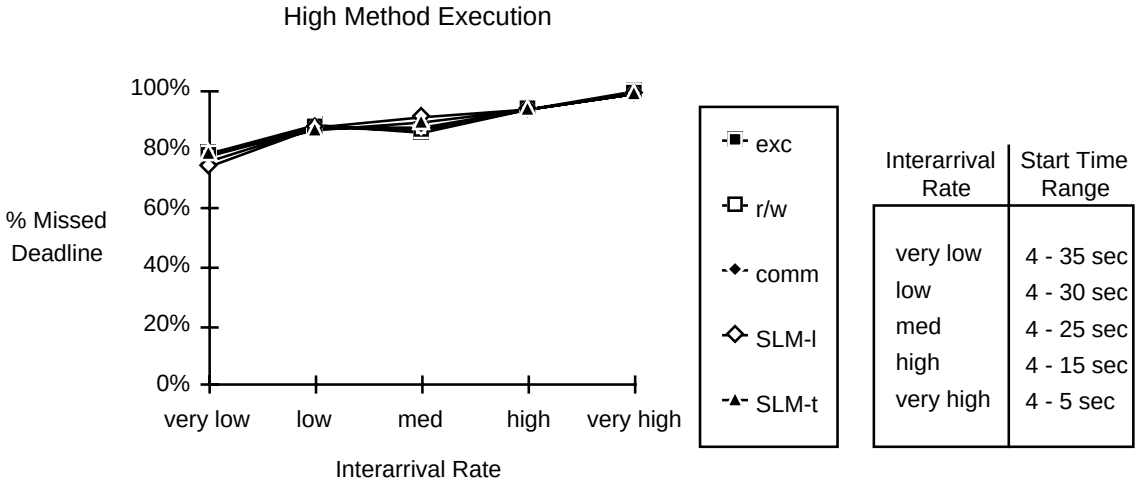

Figure 7.11: Long Methods

methods) our semantic techniques provided more flexibility for the scheduler to find them.

Test Suite DL3: Deadline. In all three of the deadline tests, the patternn were very similar. All five concurrency control mechanisms performed very much alike (within 10% of each other) at low interarrival rates. Only at the highest interarrival rate (when transactions begin within a second of each other) did we see a difference.

In the short deadline test (Figure 7.12), all of the concurrency control techniques missed around 50% of their deadlines at the very high interarrival rate. When deadlines were medium length (Figure 7.13) there was a much wider difference among the concurrency control techniques at the very high interarrival rate, with exclusive locking just below the level it was at in the short deadline test and the others spread out between 8% and 20%. In the test for long deadlines (Figure 7.14), the difference among the concurrency control techniques decreased again at the very high interarrival rate, with the semantic techniques remaining about the same as in the medium deadline tests and the other techniques performing closer to the semantic techniques. In all three tests, the semantic techniques performed almost identically.

Why did the miss ratio remain the same until the highest interarrival rate in all three tests? One reason was that given enough distance among start times, most transactions had enough time to meet short deadlines, so of course they also had enough time to meet longer deadlines. The reason that we did not see a difference between the semantic techniques in this test suite is that length of transaction deadline does not affect the temporal consistency of the data.

Test Suite DL4: Allowable Imprecision. The tests for allowable imprecision were performed only for the semantic techniques to see if there was a difference between them when varying amounts of imprecision were allowed. When no imprecision was allowed (Figure 7.15) there was virtually no difference between the techniques (when error is taken into account). Similar results were achieved with high imprecision (Figure 7.17). In this test, both techniques missed virtually no deadlines. The only difference between the two techniques in these tests can be seen in the test for medium imprecision (Figure 7.16). At low interarrival rates, the two techniques performed the same. As the interarrival rate increased, the semantic-temporal technique missed slightly fewer deadlines.

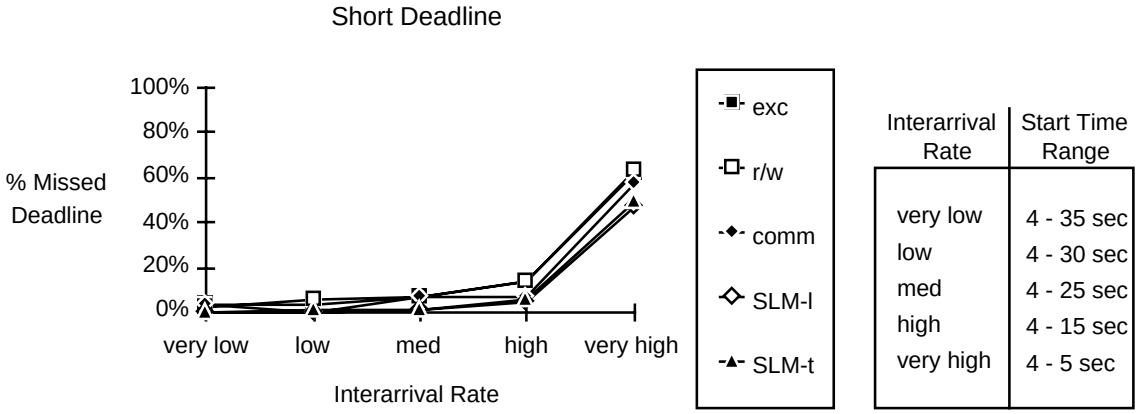

Figure 7.12: Short Deadlines

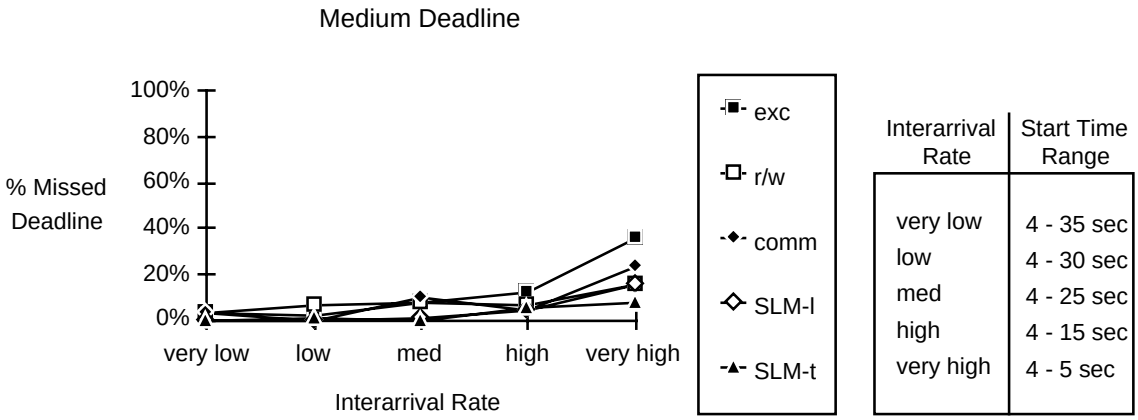

Figure 7.13: Medium Deadlines

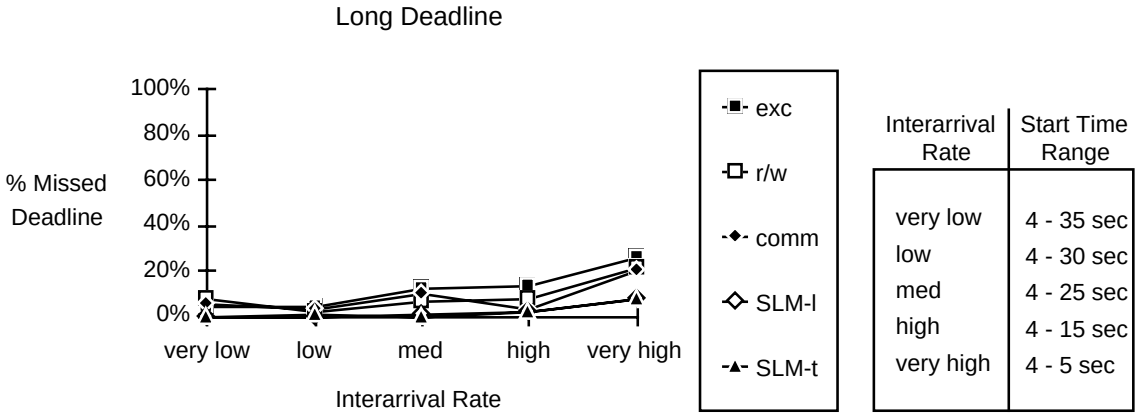

Figure 7.14: Long Deadlines

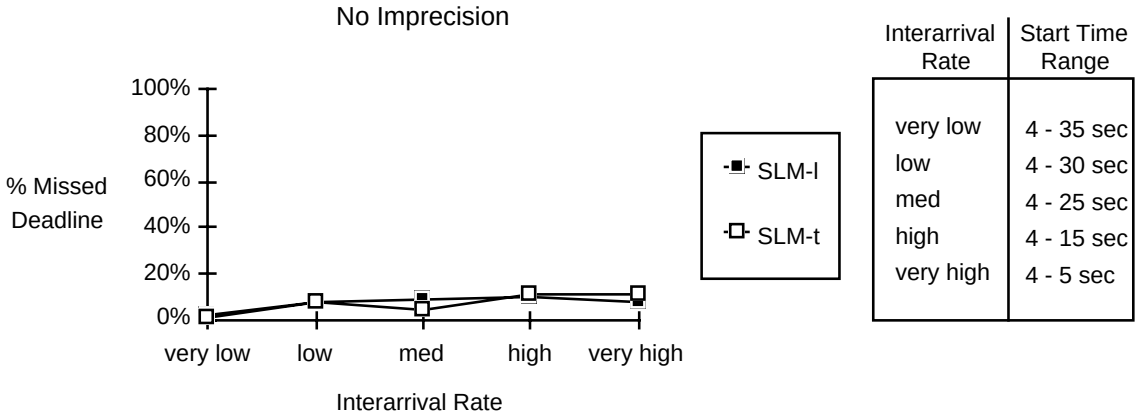

Figure 7.15: Low Imprecision

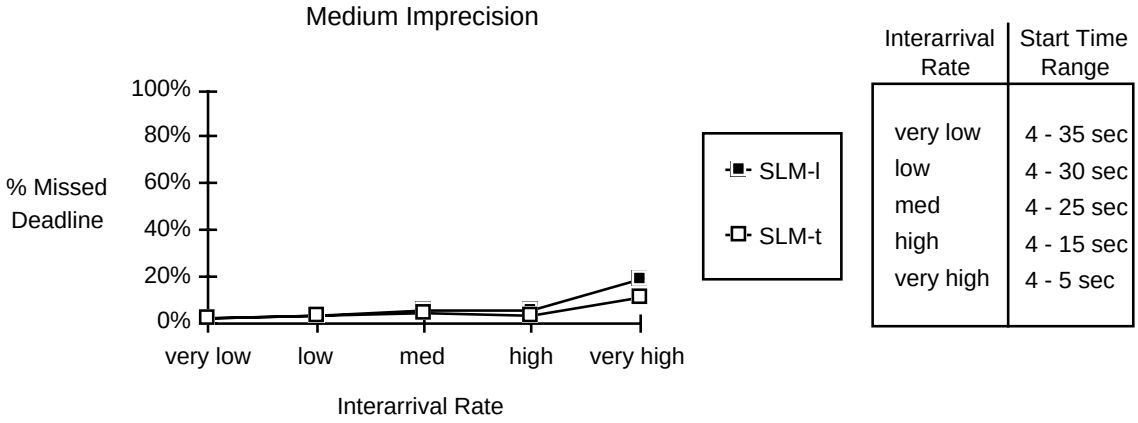

Figure 7.16: Medium Imprecision

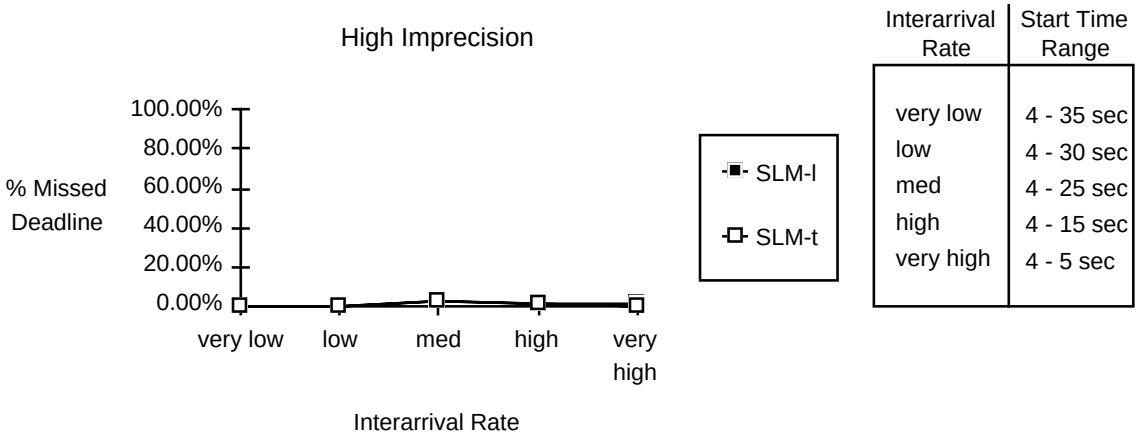

Figure 7.17: High Imprecision

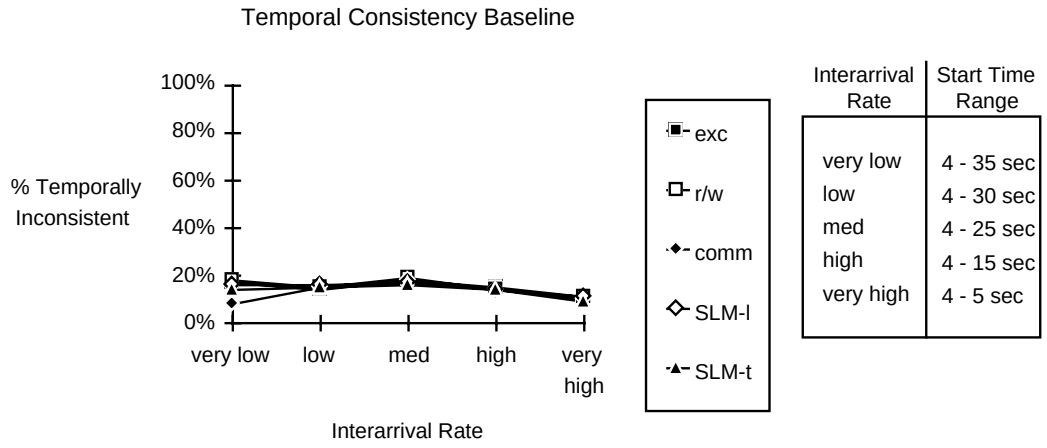

Figure 7.18: Temporal Consistency Baseline

One reason there is a slight difference between the semantic-temporal technique and the semantic-logical technique in the medium imprecision test (Figure 7.16) is that the semantic-temporal technique allowed more concurrency. We did not see a difference in the high imprecision test (Figure 7.17) because there was enough imprecision that the flexibility provided by the semantic-temporal technique was not significant. In the test involving no imprecision (Figure 7.15) we would have expected the results to be similar to the results of the test with medium imprecision because the semantic-temporal technique allowed imprecision to be introduced in the case where data temporal consistency was violated. However, there was no significant difference between the techniques.

7.7.2 Temporal Inconsistency Ratio Results

Figures 7.18-7.21 display the results of the temporal inconsistency tests. One general observation that can be made is that across the board, the temporal inconsistency was not affected by system load (interarrival rate). One explanation for this is that when the system became heavily loaded, deadlock was more likely to occur. Thus, those transactions that might have read temporally inconsistent data were caught in deadlock and did not perform the late reads. Another interesting general result was that in almost all cases, the temporal inconsistency was low (below 20%). A reason for this was that the temporal precondition (precondition **a** of Chapter 4) of the semantic locking mechanism guarded against obvious temporal inconsistency. It checked to see if an attribute might become temporally inconsistent during the execution time of the method. While the precondition did not guarantee

that transactions would not read old data, the results of these tests indicated that the it helped to keep the amount of temporally inconsistent data read low.

Test Suite TI1: Baseline Test. Figure 7.18 displays the results of the baseline temporal inconsistency test. There was no significant difference among all of the concurrency control mechanisms. This result was somewhat surprising in that we expected our semantic techniques to preserve temporal consistency better than the other object-based techniques. It may be the case that there are semantic conditions under which our techniques perform better. We first ran this test with the temporal preconditions in each of the techniques, and found no significant difference. We ran the test again with the temporal precondition removed from all but the semantic techniques to see if we could see a difference this way. The results displayed in Figure 7.18 are from the latter of the two tests.

Test Suite TI2: Method Execution. Figure 7.19 shows the results of the tests we performed to examine how length of methods affected the performance of our semantic-logical technique. The results are just as we would have expected: as the methods got longer, the temporal inconsistency ratio increased. In the tests, attributes started out temporally valid and started aging at the beginning of the test. They became fresh again when they were written. With longer methods, each access of an attribute was further from the most recent write. Thus, the attribute was more likely to be old with longer methods.

Test Suite TI3: Absolute Validity Interval. The results of the tests examining the effect of absolute validity interval on temporal inconsistency appear in Figure 7.20. For medium and high absolute validity intervals, there was virtually no temporal inconsistency. However, we see substantial temporal inconsistency with low absolute validity interval. The reason for this big difference is that the low absolute validity interval was chosen from a range of 0 to 1 seconds, and some attributes had an absolute validity interval 0 seconds. Thus, by the time the transaction read the data, it was already temporally inconsistent.

Test Suite TI4: Allowable Imprecision. Figure 7.21 shows the results of the test that was performed to illustrate how different amounts of imprecision affected temporal inconsistency. For all three tests, the semantic-logical technique had almost no temporal inconsistency. This would indicate that temporal inconsistency is not affected by imprecision

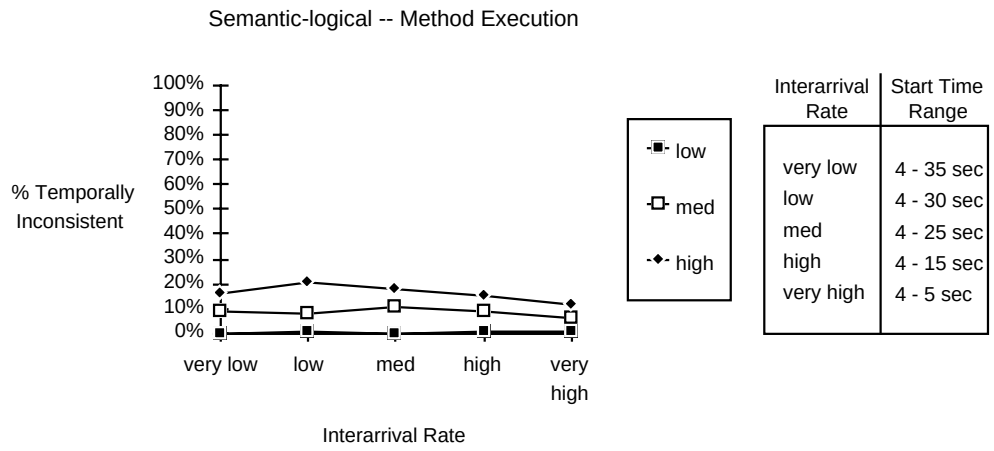

Figure 7.19: Temporal Inconsistency - Method Execution

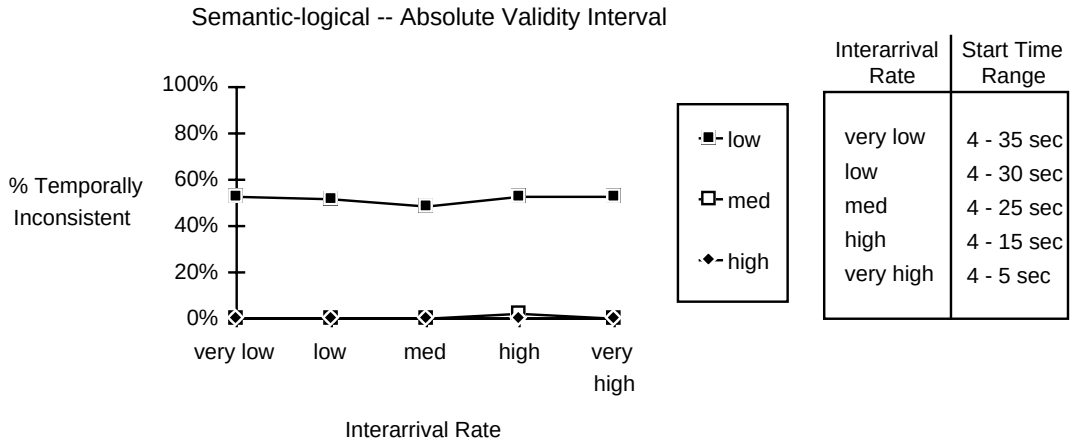

Figure 7.20: Temporal Inconsistency - Absolute Validity Interval

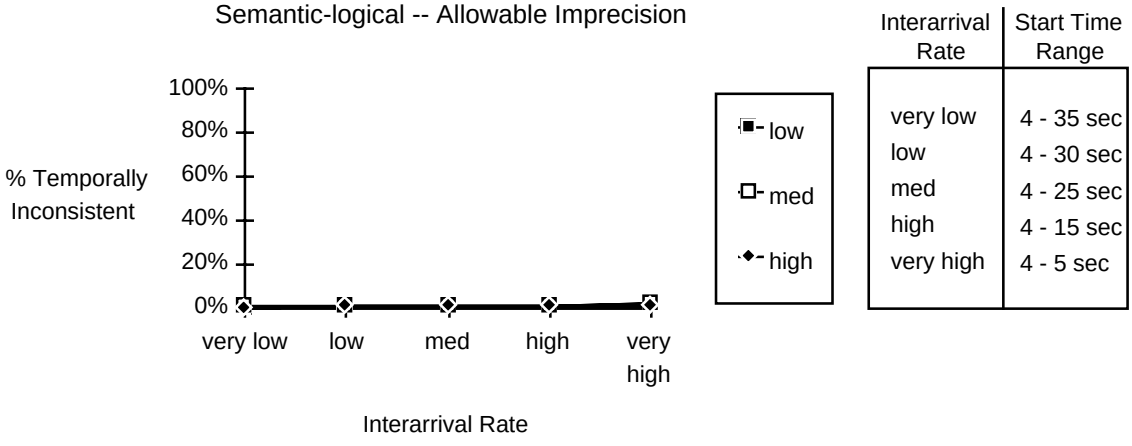

Figure 7.21: Temporal Inconsistency - Allowable Imprecision

at all. This result is contrary to the expected result, which is that more imprecision would allow reads to occur more quickly (due to higher concurrency) and thus there would be less temporal inconsistency. A possible explanation for this result is that the use of the temporal precondition takes away any significant difference we might have expected in this test.

7.7.3 Overall Results

Overall, the results of our testing were as we expected. 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. The results of the temporal inconsistency tests revealed that while there was no significant difference between our semantic techniques and the other object-based techniques, our semantic-logical technique performed best under conditions of low method execution time and attribute absolute validity interval above 1 second.

One relatively surprising result of these tests was that the amount of imprecision had little effect on missed deadline ratio or on temporal inconsistency ratio. This phenomenon requires further investigation to determine if there are semantic conditions under which imprecision does have an effect on performance. Perhaps a lower absolute validity interval or a longer method execution time would produce more expected results.

Chapter 8

Conclusion

Our goal for this work was to provide a concurrency control technique for a real-time object-oriented database that supports all four forms of consistency constraints depicted in Table 1.1 (transaction logical consistency, data logical consistency, transaction temporal consistency and data temporal consistency) and the trade-offs that result. The contributions that we have made towards reaching this goal include the definition of the RTSORAC model, the specification of the semantic locking technique, the definition of the OESR correctness criterion, and the compatibility function restrictions as sufficient conditions for maintaining OESR. We have also conducted performance tests in order to demonstrate how our technique meets our stated goal. These contributions have provided a strong foundation for future work as well. In this chapter we discuss these contributions and compare our work with the related work that was discussed in Chapter 2. We also discuss certain limitations that our work may have and point out possible future work that may alleviate these limitations.

8.1 Contributions

Recall that the goal of our work was to support the four consistency constraints: transaction logical consistency, data logical consistency, transaction temporal consistency, and data temporal consistency. The contributions enumerated above combine to provide support for all of these constraints as well as the trade-off that results.

Transaction Logical Consistency. Our semantic locking technique maintains logical consistency of transactions based on semantic consistency defined by the designer. Our

definition of an object-oriented version of the epsilon serializability correctness criterion (OESR) has provided a standard way of defining semantic consistency among transactions, while bounding imprecision. The restrictions that we specify for the definition of the compatibility function provide guidelines for the designer of the compatibility function so that OESR is maintained. As we proved in Chapter 5, the semantic locking mechanism, under the compatibility function restrictions, maintains OESR to bound imprecision.

Data Logical Consistency. The RTSORAC model provides a mechanism for specifying imprecision amounts and limits for data in objects. The compatibility function restrictions facilitate the maintenance of data logical consistency specified by the OESR correctness criterion. These features together help our semantic locking mechanism to maintain data logical consistency with bounded imprecision.

Transaction Temporal Consistency. Our semantic locking technique supports transaction temporal consistency constraints in two ways. First, the wait queue in an object is priority based. Thus, a lower priority lock request cannot be granted if there is an incompatible request in the queue with higher priority. Second, our technique supports transaction temporal consistency through the increased concurrency that it provides. Recall from Figure 1.1 that the more logically consistent schedules that a concurrency control technique allows, the more likely it is that the real-time scheduler will be able to find a schedule that is both logically consistent and temporally consistent. The flexibility of our technique, its ability to relax serializability if necessary, increases the possibility that transaction timing constraints will be met. However, because our technique provides no guarantee of predictability, it is best suited for soft real-time applications.

The performance results indicate that under many semantic conditions our technique meets as many, if not more, transaction deadlines than the other object-based techniques that we tested.

Data Temporal Consistency. Data temporal consistency constraints can be explicitly specified in the RTSORAC model. The compatibility function can use these temporal constraints to specify method interleavings that will help maintain or restore temporal consistency to the data. The temporal precondition of our semantic locking technique helps to avoid reading temporally inconsistent data. The results of our performance testing

indicate that our semantic locking technique, under various semantic conditions, allows very little temporal inconsistency (less than 20% of all method requests) to be seen by transactions.

Trade-Offs. Because we have recognized that the inherent trade-off between temporal and logical consistency exists, we have taken several measures in our work to handle it. The compatibility function of the object can specify conditions under which to sacrifice logical consistency to maintain the temporal consistency of the data. In the event that logical consistency is sacrificed, we have provided means to ensure that the semantic locking mechanism can bound the imprecision that may result: the compatibility function restrictions, the OESR correctness criterion, and the precondition tests. The results of our deadline miss ratio performance tests indicate that when there is a significant difference between the two versions of our semantic locking technique, the one that chooses temporal consistency over logical consistency (the semantic-temporal technique) preserves transaction temporal consistency better than the version that chooses logical consistency (the semantic-logical technique). This implies that if the object designer is willing to sacrifice locigal consistency, a gain can be made when temporal consistency is chosen in the trade-off.

8.2 Comparison with Related Work

In Chapter 2 we described various concurrency control techniques and pointed out why each of them did not meet the goals we set for our work. Above, we showed how our work meets our goals. Here we indicate further how our work differs from the related techniques described in Chapter 2.

Mutual Exclusion Techniques. While the mutual exclusion techniques described in Chapter 2 [DoD83, BP91, KS86] maintain consistency of data and transactions, they do not allow any concurrency among transactions. Our semantic locking technique allows as much concurrency as is allowed by the user-defined compatibility function and the transactions that access the objects. Thus, our technique can maintain mutual exclusion at a particular object by specifying no compatibility among methods in the object. And, at the same time, another object in the same database may allow more concurrency if the application requires it.

Traditional Serializability Techniques. Two-phase locking (2PL) [BHG86] is a concurrency control technique that specifies how to acquire locks. It is used along with an object-based technique that specifies what to do when conflicting locks have been requested. Traditionally, the object-based technique that is used along with 2PL is read/write locking [BHG86]. Transactions that use our semantic locking technique can acquire locks in a two-phase manner (as we did in our performance tests). Our technique provides more flexibility than read/write locking. The test results of Chapter 6 indicate that, in general, our technique, with 2PL and the semantics of our OESR restrictions, misses fewer deadlines than 2PL with read/write locking.

Several real-time concurrency control techniques based on 2PL use priority to resolve lock conflicts [AGM88, HL92]. In our semantic locking technique, if a lock request is not compatible with all currently active request and all requests in the queue with higher priority it is queued and the corresponding transaction is blocked. We have no mechanism for aborting the lower priority transaction in a conflict.

Our technique differs from the optimistic concurrency control techniques that we described [KR81, HCL90b, LS93] in one basic way, in that our technique is pessimistic. [HCL90b] uses traditional read/write locking to define conflict, and [LS93] uses timestamp intervals. It would be an interesting exercise to incorporate our compatibility function in an optimistic protocol to see how it would compare with our current technique and with other optimistic techniques.

Semantic Concurrency Control Techniques. The transaction-based semantic concurrency control techniques that we described in Chapter 2 [GM83, Lyn83, FO89, ABAK94] require the designer to define compatibility among all transaction types. Because our technique is object-based, it is more modular. That is, the designer must define compatibility among a relatively small set of object methods as opposed to a much larger set of transaction types. Also, the compatibility function restrictions described in Chapter 5 provide the designer with guidelines for defining the compatibility function to maintain bounded imprecision.

Another limitation of the transaction-based techniques that we described is the ad hoc management of imprecision. While these techniques allow concurrency that relaxes serializability, they do not provide a mechanism for handling the imprecision that may result.

Our semantic locking technique provides explicit mechanisms for specifying and bounding imprecision.

We borrowed the concept of affected sets from the work in [BR88, BR92]. Our technique is more flexible in that it can use other characteristics to determine compatibility. Also, while the techniques in [BR88, BR92, Wei88] require serializability, ours does not. The commutativity technique used in our performance tests was based on the work in [BR88]. The test results indicate that our technique allows more concurrency and provides more support for meeting temporal constraints of transactions.

The object-based technique described in [SS84] allows for non-serializable interleavings. However, it, like the transaction-based techniques, does not provide a mechanism for handling the imprecision that may result. Also, this technique, along with the object-based techniques in [BR88, BR92, Wei88] do not support temporal consistency constraints, while ours does.

Bounded Imprecision Techniques. The similarity based correctness criteria described in [KM92] provide a general way of managing imprecision. However, the similarity based technique described in [KM93] defines similarity based on the difference between timestamps on data. Our technique defines imprecision based upon differences in attribute values, not time.

Epsilon serializability [RP, DP93] provides another general correctness criterion for managing imprecision. We found it to be a useful foundation for our object-oriented ESR criterion. Our semantic locking technique differs from concurrency control techniques based upon ESR [WYP92, PHK+93] because ours allows imprecision to be introduced under special conditions, such as when temporal consistency has been violated.

Our semantic locking technique is closest to the concurrency control protocol presented in [WA92]. This protocol uses commutativity with bounded imprecision to define operation conflicts. It is similar to our technique in that the user defines the allowed amount of imprecision for a given operation invocation and the protocol uses a modified commutativity table, based on user-defined resolution set dilating functions, to determine if the operation can execute concurrently with the active operations. However, the protocol in [WA92] does not take temporal considerations into account. Furthermore, our restrictions on the compatibility function, defined in Chapter 6, provide the user with a guide for defining the

compatibility function to maintain correctness. There is no similar guide in [WA92] for defining the resolution set dilating functions used to determine compatibility.

8.3 Limitations and Future Work

The work described in this dissertation provides a strong foundation for concurrency control for a real-time object-oriented database. However, there are certain limitations to the work upon which future work can build. These limitations exist in the RTSORAC model's support for certain advanced features, in the semantic locking technique, in the support for real-time applications provided by the technique, and in the performance testing.

RTSORAC Model. Although the RTSORAC model is an object-oriented model, it does not currently support inheritance of object types or relationship types. Inheritance provides a mechanism for building type hierarchies as well as for polymorphism. The addition of inheritance to the RTSORAC model would require a careful study of how it effects all of the components of an object, including the compatibility function. For instance, if new methods are added to an object of a subclass, the compatibility function of the superclass could be inherited and then the designer would be responsible for filling in the compatibilities for the new methods with all of the inherited methods and also with each other.

Another feature of the RTSORAC model that requires more investigation is the concept of nested objects. Currently, the model allows objects to point to other objects. However, the compatibility function of the outer-most object handles concurrency for the entire nesting structure. That is, the outer-most compatibility function must be aware of the semantics of all nested objects to determine compatibility between its methods. A possible alternative to this is to coordinate the compatibility functions of all objects in the nesting. For example, the affected set of a method could contain not only the attributes of its own object, but it could also contain the affected sets of all methods in nested objects that it might call.

Semantic Locking Technique. The semantic locking mechanism currently maintains consistency for individual objects and transactions. It does not include system enforcement of inter-object concurrency control. Extending semantic locking to provide such enforcement is an area for future investigation. A possible approach could use RTSORAC relationships to

propagate locks from object to object when necessary. For instance, suppose an inter-object constraint C is defined using the method m of an object O participating in a relationship R. The corresponding enforcement rule may invoke m and therefore some measure must be taken to protect the consistency of the object O. To automatically support the inter-object constraint C, the semantic locking technique could propagate a semantic lock request through relationship R to ensure that the enforcement rule that maintains the inter-object constraint can execute while maintaining the consistency of object O.

Concurrency control and recovery have often been studied together in databases. This is because if a transaction aborts, it is important that its results do not have unwanted effects on other transactions and on the data. The work described in this dissertation does not address recovery. However, we have a strong suspicion that with the use of our OESR correctness criterion, recovery will not be necessary. That is, intuitively, as long as the amount of imprecision that would occur due to an aborted transaction does not exceed specified limits, no recovery action is necessary if the transaction aborts.

Real-Time Support. Although our semantic locking technique provides support for both transaction and data temporal consistency, there are ways in which its support for real-time applications can be improved. The technique currently requires that a semantic lock request that is not granted be blocked, regardless of its priority. Certain real-time concurrency control techniques [AGM88, HL92] use transaction priority to determine whether to block a requesting transaction or to abort a conflicting locked transaction. We could extend our semantic locking technique to perform similarly.

Another way in which we could further demonstrate support for real-time applications is to include other real-time concurrency control techniques in our performance comparison. The tests that were described in Chapter 7 involved a comparison with more traditional techniques. We did this because we found no current real-time techniques that were object-based. However, if we modify our technique to take priority into account when determining how to handle conflict, as described above, we could then perform a better comparison with real-time techniques such as [AGM88] and [HL92].

Performance Tests. The tests that we performed have provided us with valuable insight into how our semantic locking technique compares with other object-based techniques, as

well as how it performs under varying semantic conditions. The tests were also enlightening in various ways towards further studies that should be performed. The fact that the amount of allowable imprecision had no significant effect on either transaction temporal consistency or on data temporal consistency prompts further investigation. Perhaps the baseline semantics can be modified to find semantic conditions under which varying levels of imprecision will affect temporal consistency. Further testing for data temporal consistency may reveal situations in which our semantic locking technique performs better than the other object-based techniques. If not, such tests may at least reveal reasons for the results that we have found thus far.

The deadlock condition that occurred when we removed the transaction deadlines to test for data temporal consistency also requires more examination. Although the deadlock only occurred under the no deadline situation, it will be interesting to investigate how to handle deadlock that occurs because of priority. Perhaps some form of priority inheritance could avoid such a situation.

The concurrency control requirements of a real-time database are beyond those of a traditional database and have not been fully met by any current techniques that we have found. We believe that our contributions: the development of the RTSORAC model, the design of the semantic locking technique, the specification of the OESR correctness criterion, and the definition of the compatibility function restrictions provide support for the requirements of real-time databases and add to the general understanding of database concurrency control.

List of References

[ABAK94] D. Agrawal, J.L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *Proceedings of the 13th Principles of Database Systems*, pages 139–149, 1994.

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[AGM88] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *14th VLDB Conference*, August 1988.

[BHG86] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, New York, 1986.

[BOW93] Gregory A. Bussier, James Oblinger, and Victor Fay Wolfe. Real-time considerations in submarine target motion analysis. In *Proceedings of First IEEE Workshop on Real-Time Applications*, May 1993.

[BP91] Ted Baker and Offer Pazy. Real-time features for Ada 9x. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[BR88] B.R. Badrinath and Krithi Ramamritham. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37(5):541–547, May 1988.

[BR92] B.R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transaction on Database Systems*, 17(1):163–199, March 1992.

[CGM85] Ricardo Cordon and Hector Garcia-Molina. The performance of a concurrency control mechanism that exploits semantic knowledge. In *IEEE 5th International Conference on Distributed Computing Systems*, May 1985.

[CSK88] S. Cheng, J. Stankovic, and K.Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *IEEE Real-Time Systems Symposium*, pages 150–173, 1988.

[Dat86] C. J. Date. *An Introduction to Database Systems - Volume I.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[Dei84] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley Publishing Company, Reading, MA, 1984.

[DoD83] U.S. Department of Defense. Ada Programming Language, 1983. ANSI/MIL-STD-1815A-1983.

[DP93] Pamela Drew and Calton Pu. Asynchronous consistency restoration under epsilon serializability. Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[DSW90] Patrick Donohoe, Ruth Shapiro, and Nelson Weiderman. *Hartstone Benchmark User's Guide, Version 1.0.* Carnegie Mellon University, Software Engineering Institute, March 1990.

[DW93] Lisa B. Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[FO89] Abdel Aziz Farrag and M. Tamer Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database system. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[HCL90a] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1990.

[HCL90b] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. In *ACM PODS Symposium*, April 1990.

[HL92] S.L. Hung and K.Y. Lam. Locking protocols for concurrency control in real-time database systems. *SIGMOD Record*, 21(4):22–27, December 1992.

[HSTR89] Jiandong Huang, John Stankovic, D. Towsley, and Krithi Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1989.

[KM92] Tei-Wei Kuo and Aloysius K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1992.

[KM93] Tei-Wei Kuo and Aloysius K. Mok. *SSP*: A semantics-based protocol for real-time data access. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[KR81] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, Jume 1981.

[KS86] Eugene Klingerman and Alexander Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[LLS+91] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for scheduling imprecise computation. *IEEE Computer*, 24(5), May 1991.

[LS90] Yi Lin and Sang Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1990.

[LS93] Juhnyoung Lee and Sang H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[Lyn83] Nancy A. Lynch. Multilevel concurrency – a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[PDPW94] J.J. Prichard, Lisa Cingiser DiPippo, Joan Peckham, and Victor Fay Wolfe. *RTSORAC*: A real-time object-oriented database model. In *The 5th International Conference on Database and Expert Systems Applications*, Sept. 1994.

[PE94] W. Pugh and T. Marlow (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.

[PHK+93] Calton Pu, Wenwey Hseush, Gail E. Kaiser, Kun-Lung Wu, and Philip S. Yu. Distributed divergence control for epsilon serializability. In *Proceedings of 13th International Distributed Computing Conference*, June 1993.

[Raj89] R. Rajkumar. *Task Synchronization in Real-Time systems*. PhD thesis, Carnegie Mellon University, 1989.

[Ram93] Krithi Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

[RP] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. To appear in *IEEE Transactions on Knowledge and Data Engineering*. Also available as technical report No. CUCS-044-91 at Department of Computer Science, Columbia University.

[Son92] Xiaohui Song. *Data Temporal Consistency in Hard Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. Technical Report UIUCDCS-R-92-1753.

[SRSC91] Lui Sha, R. Rajkumar, Sang Son, and C. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, July 1991.

[SS84] Peter M. Schwartz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[WA92] M.H. Wong and D. Agrawal. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the 11th Principles of Database Systems*, pages 236–245, 1992.

[WBT92] David L. Wells, José A. Blakely, and Craig W. Thompson. Architechture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

[WDL93] Victor Wolfe, Susan Davidson, and Insup Lee. *RTC*: Language support for real-time concurrency. *Real-Time Systems*, 5(1):63–87, March 1993.

[Wei88] William Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[WPD+] V.F. Wolfe, J.J. Prichard, L.C. DiPippo, J. Black, J. Peckham, and P.J. Fortier. The RT-SORAC real-time object-oriented database model and prototype. *Information Systems*. submitted.

[WYP92] Kun-Lung Wu, Philip S. Yu, and Calton Pu. Divergence control for epsilon-serializability. In *Proceedings of International Conference on Data Engineering*, 1992.

[YWLS94] Philip S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

[ZM90] Stanley Zdonik and David Maier. *Readings in Object Oriented Database Systems*. Morgan Kauffman, San Mateo, CA, 1990.

Bibliography

Abbott, Robert, and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *14th VLDB Conference*, August 1988.

Agrawal, D., J.L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *Proceedings of the 13th Principles of Database Systems*, pages 139–149, 1994.

Agrawal, Rakesh, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

Badrinath, B.R., and Krithi Ramamritham. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37(5):541–547, May 1988.

Badrinath, B.R., and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transaction on Database Systems*, 17(1):163–199, March 1992.

Baker, Ted, and Offer Pazy. Real-time features for Ada 9x. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

Bernstein, Phillip A., Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, New York, 1986.

Bussier, Gregory A., James Oblinger, and Victor Fay Wolfe. Real-time considerations in submarine target motion analysis. In *Proceedings of First IEEE Workshop on Real-Time Applications*, May 1993.

Cheng, S., J. Stankovic, and K.Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *IEEE Real-Time Systems Symposium*, pages 150–173, 1988.

Cordon, Richardo, and Hector Garcia-Molina. The performance of a concurrency control mechanism that exploits semantic knowledge. In *IEEE 5th International Conference on Distributed Computing Systems*, May 1985.

Date, C. J. *An Introduction to Database Systems - Volume I*. Addison-Wesley Publishing Company, Reading, MA, 1986.

Deitel,Harvey M. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, MA, 1984.

DiPippo, Lisa B. Cingiser, and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

Donohoe,Patrick, Ruth Shapiro, and Nelson Weiderman. *Hartstone Benchmark User's Guide, Version 1.0*. Carnegie Mellon University, Software Engineering Institute, March 1990.

Drew, Pamela, and Calton Pu. Asynchronous consistency restoration under epsilon serializability. Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

Farrag, Abdel Aziz, and M. Tamer Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

Garcia-Molina, Hector. Using semantic knowledge for transaction processing in a distributed database system. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

Haritsa, Jayant R., Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. In *ACM PODS Symposium*, April 1990.

Haritsa, Jayant R., Michael J. Carey, and Miron Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1990.

Huang, Jiandong, John Stankovic, D. Towsley, and Krithi Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1989.

Hung, S.L., and K.Y. Lam. Locking protocols for concurrency control in real-time database systems. *SIGMOD Record*, 21(4):22–27, December 1992.

Klingerman, Eugene, and Alexander Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

Kung, H.T., and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, Jume 1981.

Kuo, Tei-Wei, and Aloysius K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1992.

Kuo, Tei-Wei, and Aloysius K. Mok. *SSP*: A semantics-based protocol for real-time data access. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

Lee, Juhnyoung, and Sang H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

Lin, Yi, and Sang Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1990.

Liu, J., K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for scheduling imprecise computation. *IEEE Computer*, 24(5), May 1991.

Lynch, Nancy A. Multilevel concurrency – a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

Prichard, J.J., Lisa Cingiser DiPippo, Joan Peckham, and Victor Fay Wolfe. *RTSORAC*: A real-time object-oriented database model. In *The 5th International Conference on Database and Expert Systems Applications*, Sept. 1994.

Pu, Calton, Wenwey Hseush, Gail E. Kaiser, Kun-Lung Wu, and Philip S. Yu. Distributed divergence control for epsilon serializability. In *Proceedings of 13th International Distributed Computing Conference*, June 1993.

Pugh, W., and T. Marlow (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.

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

Ramamritham, Krithi. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

Ramamritham, Krithi, and Calton Pu. A formal characterization of epsilon serializability. To appear in *IEEE Transactions on Knowledge and Data Engineering*. Also available as technical report No. CUCS-044-91 at Department of Computer Science, Columbia University.

Schwartz, Peter M., and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

Sha, Lui, R. Rajkumar, Sang Son, and C. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, July 1991.

Song, Xiaohui. *Data Temporal Consistency in Hard Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. Technical Report UIUCDCS-R-92-1753.

U.S. Department of Defense. Ada Programming Language, 1983. ANSI/MIL-STD-1815A-1983.

Weihl, William. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

Wells, David L., José A. Blakely, and Craig W. Thompson. Architechture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

Wolfe, V.F., J.J. Prichard, L.C. DiPippo, J. Black, J. Peckham, and P.J. Fortier. The RTSORAC real-time object-oriented database model and prototype. *Information Systems*. submitted.

Wolfe, Victor, Susan Davidson, and Insup Lee. *RTC*: Language support for real-time concurrency. *Real-Time Systems*, 5(1):63–87, March 1993.

Wong, M.H., and D. Agrawal. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the 11th Principles of Database Systems*, pages 236–245, 1992.

Wu, Kun-Lung, Philip S. Yu, and Calton Pu. Divergence control for epsilon-serializability. In *Proceedings of International Conference on Data Engineering*, 1992.

Yu, Philip S., Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

Zdonik, Stanley, and David Maier. *Readings in Object Oriented Database Systems*. Morgan Kauffman, San Mateo, CA, 1990.