1

THE RTSORAC REAL-TIME OBJECT-ORIENTED DATABASE PROTOTYPE

V. F. Wolfe, J.J. Prichard, L.C. DiPippo, and J. Black

Computer Science Department University of Rhode Island
Kingston, Rhode Island 02881 USA

1 INTRODUCTION

Applications that interact with their environments, such as automated vehicle control, manufacturing, and air-traffic control, have large volumes of time-constrained data on which time-constrained transactions may operate. Such applications can be supported by *real-time database management systems* [16]. Additionally, these applications often involve complex data types with complex associations among the data items. The vast majority of work on real-time databases to date has used the relational data model [16, 23], which has limitations for complex data [24].

This chapter presents a *real-time object-oriented data model* called RTSORAC (**R**eal-**T**ime **S**emantic **O**bjects **R**elationships **A**nd **C**onstraints) that incorporates time-constrained data and time-constrained transactions of real-time databases with the support for complex data provided by the object-oriented model. The chapter also presents our prototype system which is based upon the RTSORAC model. The prototype is implemented as a real-time extension to the widely-available Open Object-Oriented Database System (Open OODB) [21].

The RTSORAC model supports the expression of *temporal consistency constraints* on data. Temporal consistency constraints express how "old" data can be to still be considered valid. *Absolute temporal consistency* restricts the age of a single data item. For example, in an automated train control system, the data corresponding to a sensor that measures the speed of the train should be updated often, (e.g. every five seconds). The value of the speed is temporally consistent as long as it is no more than five seconds old. *Relative temporal* *consistency* restricts the relative ages of a group of data items with respect to each other. For example, if the train control system computes the new fuel consumption levels using the current speed and position on the tracks, it is important that the ages of the speed and position data be relatively close to one another (e.g. within two seconds) so that they represent the "same" snapshot of the environment.

RTSORAC also supports the expression of *time constrained transactions*. Timing constraints on transactions come from one of two sources. First, temporal consistency requirements of the data impose timing constraints on a transaction. For instance, the period of a sensor transaction is dictated by the absolute temporal consistency requirements of the sensor data that it writes. The second source of timing constraints on transactions is system or user requirements on reaction time. There are typically two types of timing constraints on transactions: absolute timing constraints (e.g. earliest start time, latest finish time) and periodic timing constraints (e.g. frequency of transaction initiation). The criticality of meeting timing constraints is often characterized as *hard real-time* or *soft real-time*. Although predictability is desirable, it is very difficult to achieve in a complex database system [23]. In the RTSORAC model and prototype, we concentrate on soft real-time database management, where providing predictability is desirable, but not necessary.

The addition of timing constraints on transactions and data requires advanced transaction scheduling and concurrency control techniques [1, 3, 23]. Transaction schedules should meet timing constraints and they should maintain the logical consistency of the data in the database. Most conventional database system concurrency control techniques seek to maintain logical consistency of data while not supporting temporal consistency and transaction timing constraints. For instance, a conventional technique may block a transaction with tight timing constraints or one that updates temporally invalid data if the transaction attempts to write to a data item that another transaction is reading. Alternatively, a concurrency control technique could seek to maintain temporal consistency of a data item by preempting the transaction that is reading the data item in favor of an update transaction or one with tighter timing constraints. However, this preemption may violate the logical consistency of the data or the logical consistency of the preempted reading transaction. The RTSORAC model supports expression of both logical and temporal consistency requirements and their trade-offs, as described in Section 2. Our prototype system uses a novel *real-time semantic concurrency control* technique to enforce these constraints and trade-offs, as described in Section 3.

In situations where logical consistency is traded for temporal consistency, *imprecision* may be introduced into a transaction's "view" of the data or into the data value itself. Many real-time control applications allow a certain amount of imprecision . For instance, it may be sufficient for the stored data value representing a train's oil pressure to be within a specified number of units of an exact value. Also, since much of the data in real-time applications is periodically read from sensors, allowing temporary imprecision may be permissible because precise values can be restored on the next update. However, even if imprecision is allowed, it typically must be bounded in the system. The RTSORAC model of Section 2 provides for the expression of imprecision accumulation and bounds; Section 3 describes how the prototype system's concurrency control technique enforces the imprecision bounds.

Other work has been done on temporal consistency enforcement, real-time transaction management, and imprecision in real-time databases (see [23] for a survey). This work has primarily involved extensions to the relational data model. Although the relational model is useful for many applications, there are several reasons why we believe that it is not as well-suited as an *object-oriented database model* (OODM) (for a survey of object-oriented database research see [24]) for many other real-time applications. An OODM allows for the specification of more complex data types than those typically allowed in relational databases. The encapsulation mechanisms of a OODM allow constraints that are specific to a data object to be enforced within the object. That is, instead of imposing a correctness criterion that ignores temporal consistency, such as serializability, the schema designer can express both logical and temporal consistency constraints for each individual object. This allows for more flexible correctness criteria to be used. The capability to include user-defined operations (methods) on data objects can improve real-time capabilities by providing complex operations with well-known timing behavior and by allowing a wide range of operation granularities for semantic real-time concurrency control. That is, instead of only enforcing concurrency among read and write operations, as is typically done in relational data models, the OODM can potentially allow for the enforcement of concurrency among a rich set of user-defined operations on objects.

The remainder of this chapter is structured as follows. Section 2 describes the RTSORAC real-time object-oriented data model. Section 3 summarizes our prototype development which includes techniques for real-time concurrency control, real-time scheduling, and data definition/data manipulation under the RTSORAC model. Section 4 reviews the strengths, weaknesses, and current work involving the model and implementation.

2 THE RTSORAC MODEL

The RTSORAC model incorporates features that support the requirements of a real-time database into an extended object-oriented model. It has three components that model the properties of a real-time object-oriented database: *objects*, *relationships* and *transactions*. Objects represent database entities. RTSORAC extends a traditional object model with attributes that have *time* and *imprecision* fields. Objects are also extended to express *constraints*: logical constraints (on the value fields of attributes), temporal constraints (on the time fields of attributes), and bounds on imprecision (on the imprecision fields of attributes). To support trade-offs among conflicting constraints, each object also expresses a compatibility function among its methods. RTSORAC relationships represent associations among the database objects. Relationships also express inter-object constraints. RTSORAC transactions access the objects and relationships in the database. These transactions can have timing and imprecision constraints.

We now describe each of the RTSORAC model components in detail.

2.1 Objects

An *object* (Figure 1) consists of five components, $\langle N, A, M, C, CF \rangle$, where N is a unique name or identifier, A is a set of attributes, M is a set of methods, C is a set of constraints, and CF is a compatibility function. Figure 2 illustrates an example of a **Train** object (adapted from [4]) for storing information about a train control system in a database.

$$\begin{aligned}
\textbf{Object} &= \langle N, A, M, C, CF \rangle \\
N &= UniqueID \\
A &= \{a_1, a_2, ..., a_m\} \text{ where attribute } a_i = \langle N_a, V, T, I \rangle \\
M &= \{m_1, m_2, ..., m_n\} \text{ where method } m_i = \langle N_m, Arg, Exc, Op, OC \rangle \\
C &= \{c_1, c_2, ..., c_s\} \text{ where constraint } c_i = \langle N_c, AttrSet, Pred, ER \rangle \\
CF &= \text{compatibility function}
\end{aligned}$$

Figure 1 Object characteristics in RTSORAC

Attributes.

Each attribute of an object is characterized by $\langle N_a, V, T, I \rangle$. N_a is the name of the attribute. The second field, V, is used to store the value of the attribute, and may be of some abstract data type. The next field, T, is used to store the time at which the value was recorded. Access to the time field of an attribute is necessary for maintaining the attribute's temporal consistency. For example, in the **Train** object, there is an attribute for storing the oil pressure called `OilPr` which is updated periodically by a sensor. This update is expected every thirty seconds, thus the `OilPr` attribute is considered temporally inconsistent if the update does not occur within that time frame. The system must examine the time field of the `OilPr` attribute to determine if the update occurs as expected.

The last field, I, of an attribute is used to store the amount of imprecision associated with the attribute. This field is of the same type as the value field V. We elaborate on the management of imprecision in our discussion of an object's compatibility function later in this section, and in Section 3.4.

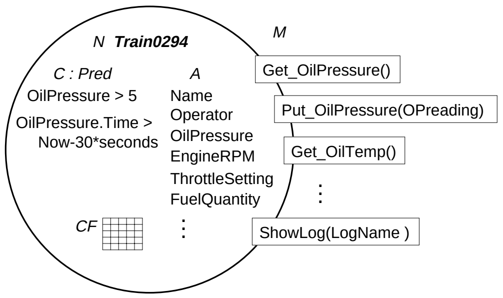

Figure 2 Example of **Train** object

Methods.

Each method of an object is of the form $\langle N_m, Arg, Exc, Op, OC\rangle$. N_m is the name of the method. Arg is a set of arguments for the method, where each argument has the same components as an attribute, and is used to pass information in and/or out of the method. Exc is a set of exceptions that may be raised by the method to signal that the method has terminated abnormally.

Op is a set of operations that represent the actions of the method. These operations include statements for conditional branching, looping, I/O, and reads and writes to an attribute's value, time, and imprecision fields.

The last characteristic of a method, OC, is a set of operation constraints. An operation constraint is of the form $\langle N_{oc}, OpSet, Pred, ER\rangle$ where N_{oc} is the name of the operation constraint, $OpSet$ is a subset of the operations in Op, *Pred* is a predicate (Boolean expression), and *ER* is an enforcement rule. The predicate is specified over *OpSet* to express precedence constraints, execution constraints, and timing constraints [22]. The enforcement rule is used to express the action to take if the predicate evaluates to false. A more complete description of an enforcement rule can be found in the paragraphs below describing constraints.

Here is an example of an operation constraint predicate in the **Train** object:

Pred : `complete(Put_OilPr) < NOW + 5*seconds`

A deadline of `NOW + 5*seconds` has been specified for the completion of the `Put_OilPr` method. Note the use of a special atom `complete(e)`, which represents the completion time of the executable entity `e`. Other atoms that are useful in the expression of timing constraints include `start(e)`, `wcet(e)`, and `request(e)` which represent the execution start time, worst case execution time, and the execution request time of entity `e` respectively.

Constraints.

The constraints of an object permit the specification of correct object state. Each constraint is of the form $\langle N_c, AttrSet, Pred, ER\rangle$. N_c is the name of the constraint. *AttrSet* is a subset of attributes of the object. *Pred* is a predicate that is specified using attributes from the *AttrSet*. The predicate can be used to express the logical consistency requirements by using value fields of the attributes. It can express temporal consistency requirements by using the time fields of attributes. It can express imprecision limits by using the imprecision fields of attributes.

The enforcement rule (*ER*) is executed when the predicate evaluates to false, and is of the form $\langle Exc, Op, OC\rangle$. *Exc* is a set of exceptions that the enforcement rule may signal, *Op* is a set of operations that represent the actions of the enforcement rule, and *OC* is a set of operation constraints on the execution of the enforcement rule.

As an example of a temporal consistency constraint, consider the following. As mentioned earlier, the **Train** object has an oil pressure attribute that is updated with the latest sensor reading every thirty seconds. To maintain the temporal consistency of this attribute, the following constraint is defined:

```
N :        OilPr_avi
AttrSet : {OilPr}
Pred :     OilPr.time > Now - 30*seconds
ER :       if Missed <= 2 then
             OilPr.time = Now
             Missed = Missed + 1
             signal OilPr_Warning
           else signal OilPr_Alert
```

The enforcement rule specifies that if only one or two of the readings have been missed, a counter is incremented indicating that a reading has been missed and a warning is signaled using the exception `OilPr_Warning`. If more than two readings have been missed, then an exception `OilPr_Alert` is signaled, which might lead to a message being sent to the train operator. The counter `Missed` is reset to zero whenever a new sensor reading is written to the `OilPr` attribute.

Compatibility Function.

The compatibility function of an object expresses the semantics of allowable concurrent execution of each ordered pair of methods in the object. For each ordered pair of methods, (m_i, m_j), a Boolean expression $(BE_{i,j})$ is defined. $BE_{i,j}$ can be evaluated to determine whether or not m_i and m_j can execute concurrently. In many object-oriented systems, the execution of a single method of an object prevents any other methods of the object from being executed, i.e. the entire object is locked upon invocation of a single method. Through the use of the compatibility function, the designer of an object can allow more flexibility in sharing of objects by defining the semantics of the compatibility of each pair of methods.

In the ordered pair (m_i, m_j) for which $BE_{i,j}$ is defined, m_i represents a method that has an active invocation, and m_j represents a method for which an invocation has been requested by a transaction. The boolean expression may involve predicates for several system characteristics including: affected sets [5], which are the attributes of the object that can be read or written by a method; the current time and the temporal consistency requirements of attributes; the current amounts and limits of imprecision of attributes and method arguments; the object's other active methods, as well as other characteristics [8]. All of the information that the compatibility function uses to make its determination is available locally within the object or in the arguments of the methods involved. If the compatibility function evaluates to TRUE, then the method invocations may execute concurrently; otherwise, they should not be allowed to execute concurrently.

	Compatibility	**Imprecision Accumulation**
A:	$CF(Put\_OilPr(), Put\_OilTemp()) = TRUE$	No Imprecision
B:	$CF(Get\_OilPr(P_1), Put\_OilPr(P_2)) = (OilPr.time <= Now - 30 * sec)\ AND\ (\|OilPr.value - P_2.value\| <= (P_1.implimit - P_1.ImpAmt)$	Increment $P_1.ImpAmt$ by $\|OilPr.value - P_2.value\|$
C:	$CF(Put\_OilPr(P_1), Put\_OilPr(P_2)) = (\|P_1.value - P_2.value\| <= OilPr.implimit - OilPr.ImpAmt$	Increment $OilPr.ImpAmt$ by $\|P_1.value - P_2.value\|$

Figure 3 Compatibility Function Examples

Based on the semantics of the application, the compatibility function may allow method interleavings that introduce imprecision into the attributes and method arguments. Therefore, in addition to specifying compatibility between two method invocations, the compatibility function expresses information about the potential imprecision that could be introduced by interleaving method invocations. There are three potential sources of imprecision when methods invocations m_i and m_j are interleaved: imprecision in the value of an attribute that is written by both m_i and m_j, imprecision in the value of the return arguments of m_i when m_i reads attributes written by m_j and imprecision in the value of the return arguments of m_j when m_j reads attributes written by m_i [17].

Figure 3 demonstrates several examples of the compatibility function and its associated imprecision accumulation for the **Train** object of Figure 2. In Example **A** of Figure 3, the compatibility function is used to specify that the methods `Put_OilPr` and `Put_OilTemp` can always run concurrently. This is appropriate because these two methods access different attributes. No imprecision is introduced in this case. Example **B** demonstrates trading off logical consistency for temporal consistency. If the temporal consistency constraint on the `OilPr` attribute has been violated ($OilPr.time <= Now - 30 * seconds$), then the compatibility function specifies that the `Put_OilPr` method invocation can execute concurrently with an active `Get_OilPr` method, presumably to restore the temporal consistency of the `OilPr` attribute. The CF restricts this interleaving to occur only if the amount of imprecision in the argument P_1

	Relationship	$= \langle N, A, M, C, CF, P, IC\rangle$
N	$=$	$UniqueID$
A	$=$	$\{a_1, a_2, ..., a_m\}$ where attribute $a_i = \langle N_a, V, T, I\rangle$
M	$=$	$\{m_1, m_2, ..., m_n\}$ where method $m_i = \langle N_m, Arg, Exc, Op, OC\rangle$
C	$=$	$\{c_1, c_2, ..., c_r\}$ where constraint $c_i = \langle N_c, AttrSet, Pred, ER\rangle$
CF	$=$	compatibility function
P	$=$	$\{p_1, p_2, ..., p_s\}$ where participant $p_i = \langle N_p, OT, Card\rangle$
IC	$=$	$\{ic_1, ic_2, ..., ic_t\}$ where interobject constraint $ic_i = \langle N_{ic}, PartSet, Pred, ER\rangle$

Figure 4 Relationship Characteristics in RTSORAC

of the `Get_OilPr` method invocation does not exceed the limit specified by the invoking transaction (P_1.*implimit*, see Section 2.3). The amount of imprecision to add to P_1 in this case is also specified by the compatibility function. Example **C** demonstrates how the `OilPr` attribute can become imprecise if two sensor transactions individually invoke the `Put_OilPr` method and these methods are allowed to interleave. Note that although we use only simple methods (essentially reads and writes) in this example, the compatibility function can specify imprecision accumulation for general object methods [8].

2.2 Relationships

Relationships represent aggregations of two or more objects. In the RTSORAC model, a *relationship* (Figure 4) 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. In addition, objects that can participate in the relationship are specified in the participant set P, and a set of interobject constraints is specified in IC.

Figure 5 illustrates an example of an **Energy Management** relationship for relating a **Train** object with a **Track** object. The **Track** object stores information such as track profile and grade, speed limits, maximum load, and power available. The **Energy Management** relationship uses both train and track information to determine control algorithm parameters such as fuel efficient throttle and brake settings.

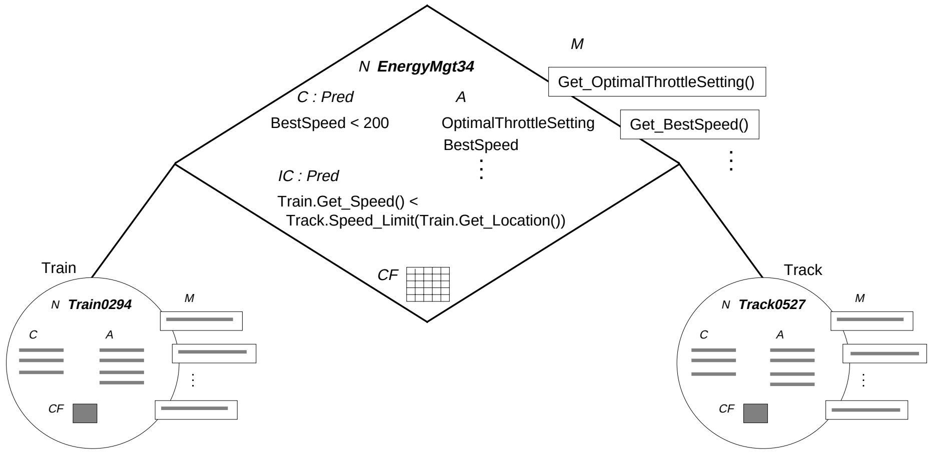

Figure 5 Example of **Energy Management** relationship

Participants.

Each participant in a relationship is of the form $\langle N_p, OT, Card\rangle$. N_p is the name of the participant. OT is the type of the object participating in the relationship. $Card$ is the cardinality of the participant, which is either *single* or *multi* [7]. Constraints can be used to express cardinality requirements of the relationship, such as minimum and maximum cardinality of the participants. In Figure 5, `Train` and `Track` are single cardinality participants.

Interobject Constraints.

The interobject constraints placed on objects in the participant set are of the form:
$\langle N_{ic}, PartSet, Pred, ER\rangle$. N_{ic}, $Pred$, 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 by $\langle Exc, Op, OC\rangle$, however the operations in Op can now include invocations of methods of the objects participating in the relationship.

As an example of an interobject constraint, consider the **Energy Management** relationship in Figure 5. A **Train** object will be on one specific segment of track, represented by the **Track** object participating in the relationship. The train should obey the speed limits set on the track segment, so the following interobject constraint predicate could be specified:

$Pred:$ `Train.Get_Speed() < Track.Speed_Limit(Train.Get_Location())`

If the speed of the train exceeds the speed limit posted at the train's location on the track, the corresponding enforcement rule signals `SpeedLimitExceeded`.

2.3 Transactions

A *transaction* has six components, $\langle N_t, O, OC, PreCond, PostCond, Result\rangle$, where N_t is a unique name or identifier, O is a set of operations, OC is a set of operation constraints, $PreCond$ is a precondition, $PostCond$ is a postcondition, and $Result$ is the result of the transaction. Each of these components is briefly described below.

Operations.

The operations in O represent the actions of the transaction. They include statements of the language in which the transaction is written, and method invocations on database objects (MI). Method invocations (MI) are of the form $\langle MN, ArgInfo\rangle$, where MN is the method name (prepended with the appropriate object identifier) and $ArgInfo$ is a set of tuples containing argument information. Each argument tuple is of the form $\langle aa, maximp, tcr\rangle$ where aa is the actual argument to the method, $maximp$ is the maximum allowable imprecision of the argument, and tcr is the temporal consistency requirement of the argument. The fields $maximp$ and tcr are specified only for arguments that are used to return information to the transaction. These fields allow the transaction to specify requirements that differ from those defined on the data in the objects. For example, the transaction might be willing to accept a value whose temporal consistency requirements have been violated so as to meet other timing constraints. The data may still be useful to the transaction because of other available information (for example, it may be able to do some extrapolation). A transaction may also specify that data returned by a method invocation must be precise ($maximp$ is zero).

Operation Constraints.

OC is a set of constraints on operations of the transaction. These constraints are of the same form as the operation constraints specified for methods, $\langle N_c$, *OpSet*, *Pred*, *ER*$\rangle$. They can be used to express precedence constraints, execution constraints, and timing constraints. For example, a transaction may require that a sensor reading be returned within two seconds.

Precondition, Postcondition, Result.

PreCond represents preconditions that must be satisfied before a transaction is made ready for execution. For example, it may be appropriate for a transaction to execute only if some specified event has occurred. The event may be the successful termination of another transaction, or a given clock time. *PostCond* represents postconditions that must be satisfied upon completion of the operations of the transaction. The postconditions can be used to specify the semantics of what constitutes a *commit* of a transaction containing subtransactions. *Result* represents information that is returned by the transaction. This may include values read from objects as well as values computed by the transaction.

3 IMPLEMENTATION OF THE RTSORAC MODEL

To implement the RTSORAC model in a prototype system, we have extended the Open Object-Oriented Database System (Open OODB) [21]. The open, modular design of Open OODB facilitates extending it with features to support specification and management of RTSORAC objects, relationships, and transactions. The following sections summarize the Open OODB system, describe interface extensions to Open OODB, and discuss the extensions to the Open OODB architecture. The interface extensions involve a graphic interface for specifying classes for RTSORAC objects. Extensions to the Open OODB architecture include real-time transaction management that performs earliest-deadline-first scheduling and real-time object management for shared-memory RTSORAC objects.

3.1 Open Object-Oriented Database System

The Open OODB system was initiated by the U.S. Advanced Research Projects Agency (ARPA). An alpha version was released in 1993 and subsequent versions have been released in 1996. The project's goal is to establish a common, modular, modifiable, object-oriented database system suitable to be used by a wide range of researchers and developers [21]. Open OODB is designed so that features such as transaction management, query interface, persistence, etc. are modules that can be individually "unplugged" and replaced by other modules.

Open OODB's computational model strives to transparently extend the behavior of objects that are found in application programming languages. The current release is a transparent extension to C++. In Open OODB's computational model, objects can exist in one of many address spaces. Currently there are two address spaces supported: transient, which resides in main memory, and persistent, which resides remotely in the Exodus [6] storage manager. The system provides communication and translation facilities to allow transfers between different address spaces to an Open OODB transaction's address space. That is, an Open OODB transaction that wishes to use an object is granted a lock on that object and the Open OODB run-time system copies the object into the transaction's address space. There are language extensions to C++ that specify requests for objects as well as extensions that specify other database functionality.

The basic conceptual system architecture of Open OODB is shown in Figure 6 (along with the proposed real-time extensions that we have added). The *support managers* are modules that are currently implemented as library routines that get linked into the user's C++ program to (transparently) provide the extended database capabilities. The *Address Space Manager* supports mappings between global identifiers and object identifiers used in the local address space. The *Communication Manager* provides support for interfacing to one or more underlying communications mechanisms. The *Translation Manager* translates an object stored in one format to a target format. For instance, it translates objects stored in Exodus into objects suitable for a C++ application program. The *Data Dictionary* is a globally known repository of the data model and type information, instance information, name mappings (of application names to instances) and possibly system configuration and resource utilization information.

Policy managers (PMs) provide extenders to the behavior of programs by coordinating the support managers just described. The *Persistence Policy Manager*

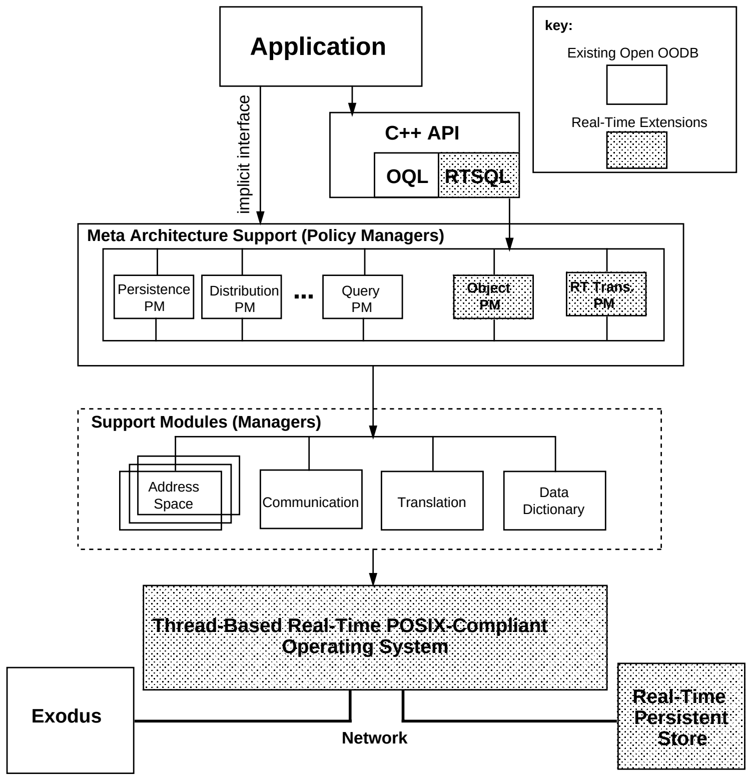

Figure 6 Open OODB Architecture With Real-Time Extensions

provides applications with an interface through which they can create, access, and manipulate persistent objects in various address spaces. The *Transaction Policy Manager* enables concurrent access to persistent and transient data; its implementation in the current release is a trivial mapping to Exodus write locks on all objects. Other policy managers include those for distribution, change management, indexing, and query processing.

The query interface is in two forms: an extended version of C++ and an SQL-like language called OQL, which must be embedded in C++ code [21]. The C++ interface is C++ code extended with methods that invoke capabilities of the managers. OQL has a very basic set of SQL-like commands that work on sets of objects. Although the current version is skeletal, actual examples can be executed. It relies heavily on Exodus as its persistent storage and for concurrency control and recovery. Eventually, as Open OODB development progresses, many of the manager capabilities will be incorporated into the Open OODB architecture (Figure 6).

3.2 RTSORAC Extensions to the Open OODB Interface

In the current version of Open OODB, the schema is specified as a collection of C++ classes and transactions are specified as C++ programs, or as OQL programs that are compiled to C++ programs. Recall that objects and transactions in the RTSORAC model have additional features beyond those supplied by C++ classes and programs. To support these additional capabilities, we have added two additional interfaces to Open OODB: a graphic interface to specify classes for RTSORAC objects, and real-time extensions to the standard SQL query language to specify RTSORAC transactions.

Graphic RTSORAC Schema Specification. A schema in our extended Open OODB prototype is specified as classes for RTSORAC objects which are translated to C++ code suitable to execute on the extended Open OODB system. Specification of these classes is done with a graphic interface programmed with X-windows and Motif. The graphic interface directs database schema specification while following Rumbaugh's OMT model [12] for object-oriented design with additional real-time features [20]. The interface is a version of a Motif-based software package called *Object-Oriented Designer* from Pusan University, Korea, extended to provide facilities to define the RTSORAC model features that must be added to C++ classes.

Our prototype implementation provides attributes having only floating point value fields. We have designed the capability to create arbitrary real-time attribute classes by using the type of the value field as an argument to a C++ template that provides time and imprecision capabilities. The compatibility function and constraints are added to classes through a special tool [11] that can be selected from the graphic interface menu. The tool parses the specification of a schema object and computes a default compatibility function based on affected sets of methods (see Section 2.1). It then interacts with the schema designer to incorporate semantic information into the compatibility function.

The interface produces a C++ class specification with certain "meta members", including a wait queue, compatibility function, POSIX mutual exclusion locks (mutexes) and condition variables, and member functions to request and release locks on the object. These meta members are used by the concurrency control mechanism described later in Section 3.3.2. The compatibility function defined by the interface tool is structured as a two dimensional array. The rows represent currently active methods and the columns represent requested methods. Each element of the array is a TRUE, FALSE, or pointer to a user-defined boolean function. The array element determines whether the methods which correspond to the element's row and column, are compatible. For instance, the array representing the compatibility function of a **Train** object would have a pointer to a boolean function specifying each of the conditions and associated imprecision accumulation shown in Figure 3 of Section 2.1.

Transactions. Transactions in our prototype are C++ programs that include the schema file of object type declarations which was generated by the graphical interface. Each transaction program is compiled into a POSIX process that maps all database objects, which reside in shared memory as described in Section 3.3.2, into its own address space. The process uses calls to the concurrency control mechanism (Section 3.3.2) to lock objects while using them. These calls are provided by our Open OODB policy manager code, as shown in Figure 7 of Section 3.3.2. Once an 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.

3.3 RTSORAC Extensions to the Open OODB Architecture

Our RTSORAC extensions to the Open OODB architecture are designed within Open OODB's original framework, as shown in Figure 6. We have made two changes to the system's underlying architecture by implementing extensions using a real-time POSIX operating system [15] and by incorporating a real-time persistent storage subsystem. We have also added a policy manager for real-time transaction management and a policy manager for real-time object management.

Basic Open OODB Architecture Modifications

As shown in Figure 6, our prototyping uses a real-time operating system that is consistent with the POSIX standards and a real-time persistent storage manager. The current release of Open OODB executes on a Sun Sparc architecture with the Sun Solaris operating system. Solaris contains many of the real-time operating system features specified in the IEEE/ISO POSIX real-time operating system standards [15]. These features include shared memory, priority-based scheduling and priority-based semaphores.

Our major basic architecture modification is the addition of a real-time persistent store. The current Open OODB version relies heavily on the Exodus storage manager as its persistent store. Exodus's unpredictable execution times, handling of requests in first-come-first-serve order rather than priority order, and conservative locking capabilities, render it unacceptable for a real-time data management system. Instead of relying on Exodus, we are incorporating another address space to Open OODB: a real-time persistent address space. Our current design uses this address space as checkpointed permanent storage for shared main memory RTSORAC objects (see Section 3.3.2) and for swap space if all objects can not fit into shared memory.

RTSORAC Object Management

RTSORAC database objects are designed to be kept in shared main memory for fast, predictable access. That is, instead of keeping objects in one of the current Open OODB address spaces, where they must be copied into a transaction's local address space for use, the protoype system keeps objects in shared main memory. The *Object Policy Manager* (OPM) that we have added to Open

OODB manages this shared memory and provides concurrency control for the objects. Figure 7 shows the implementation of object management.

Shared Main 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 7), 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 C++ `delete` operator for removing objects.

Semantic Locking Object Concurrency Control. Since each transaction may concurrently map objects in the shared memory segment into its own virtual address space, we must provide a concurrency control mechanism for the shared objects. Open OODB's current policy enforces serializability through exclusive locking of objects by transactions before a transaction makes a copy of the object into it's own address space. This is quite slow compared to shared memory accesses. Additionally, the exclusive locking of objects ignores transaction and data timing constraints.

We have developed a concurrency control technique called *semantic locking* for RTSORAC object management [8]. The semantic locking technique is capable of supporting logical consistency, temporal consistency, and the trade-offs between them as well as bounding any resulting imprecision. The technique utilizes the user-defined compatibility function (Section 2.1) of a RTSORAC object to determine the trade-off and to define correctness for that particular object. In this technique, a transaction requests a semantic lock to invoke a method on an object. Semantic locks are granted based on the evaluation of a set of conditions and on the evaluation the compatibility function of the object.

When a transaction requests a semantic lock for a method invocation, it calls the meta member function `SLM_lock()` of the object 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 `SLM_lock` meta member function attempts to acquire a semantic lock for the transaction. There are two possible outcomes when a transaction process requests a semantic lock for a method invocation: the `SLM_lock` meta member function either grants permission to the transaction process to execute the requested method, or it suspends the requesting transaction. A suspended transaction will be awakened and will retry its lock request whenever a lock is released (discussed later). In either case, the transaction releases the mutex at the end of the `SLM_lock` meta member function. Note that the OPM uses mutexes 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.

Figure 8 shows the semantic locking mechanism that the `SLM_lock` meta member function performs when a transaction requests a semantic lock for a method invocation m_{req}. First, `SLM_lock` computes the maximum amount of imprecision that m_{req} could introduce into each of the attributes that it writes and into each of its own return arguments (Step A). It computes these values by using the amount of imprecision already in the attribute or return argument and calculating how m_{req} may update this imprecision through operations that it performs.

Next, the meta member function evaluates a set of conditional statements that determine if granting the lock would violate temporal or imprecision constraints. The first condition ensures that if a transaction requires temporally valid data, then m_{req} will not execute if any of the data that it reads will become temporally invalid during its execution time. The other two conditions test that m_{req} will not introduce too much imprecision into the attributes that it writes and into its return arguments.

If all of the above conditions hold, the `SLM_lock` meta member function updates the imprecision amounts computed in Step A and saves the old amounts in a data structure, in case the request is not granted (Step C). The meta member function then loops to evaluate the compatibility function for m_{req} with each currently locked method invocation and with each lock request in the wait queue for a method invocation with higher priority than m_{req} (Step D). If all tests in the loop succeed, the meta member function grants the lock for m_{req}, adds it to the active locks set and gives the transaction permission to execute the method. If any of the conditions or any compatibility test fails, the `SLM_lock`

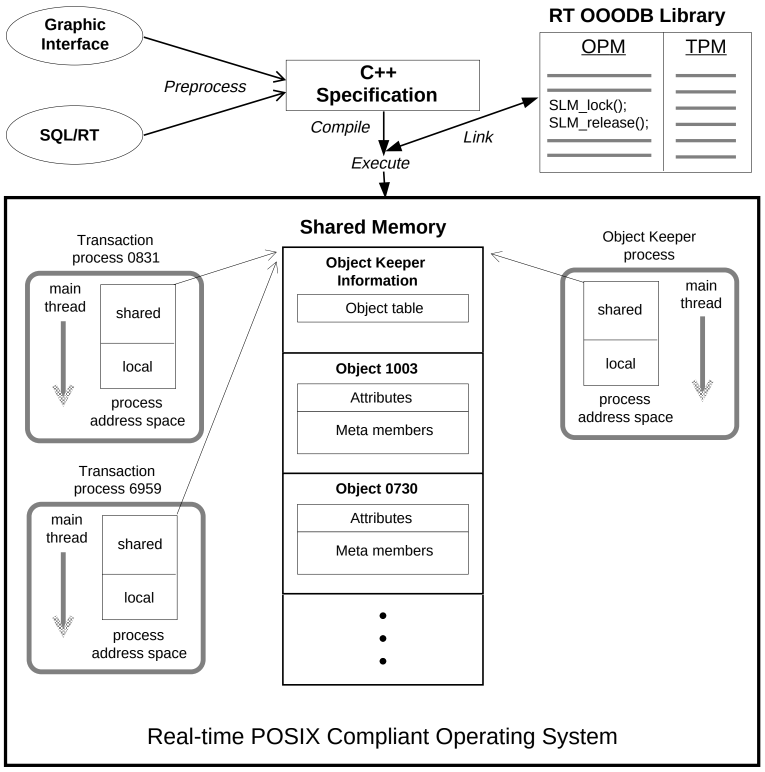

Figure 7 Object Management Implementation in Open OODB

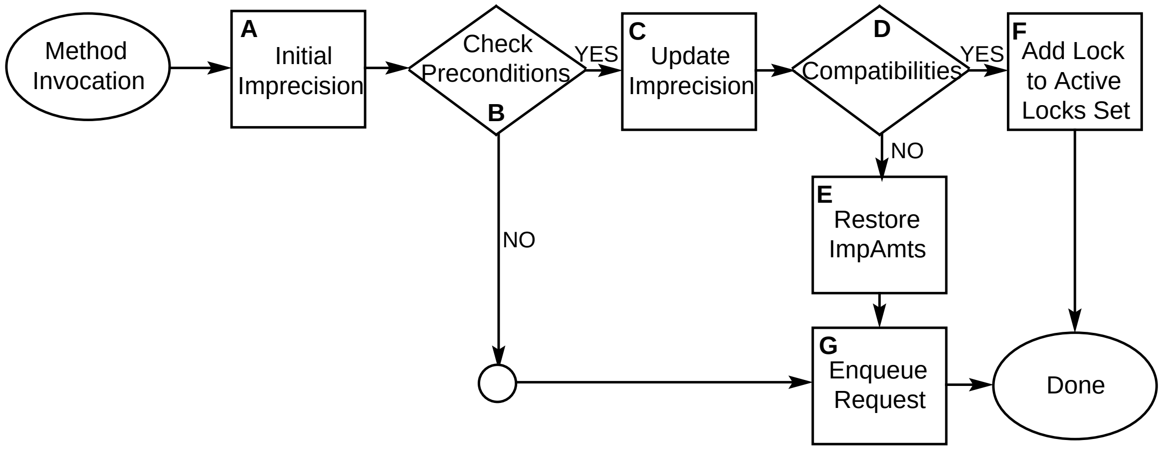

Figure 8 SLM_lock Meta Member Function Outline

meta member function restores the original values of any changed imprecision amounts (Step E), places the lock request in the priority queue, and suspends the requesting transaction (Step G).

A transaction must explicitly release the locks that it is granted by calling the `SLM_release` 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 so they may retry their lock requests. Due to the newly-released lock, it may now be possible to grant some of these previously-denied locks. The use of a the real-time POSIX scheduler, discussed next, assures that the awakened transactions make their lock requests in priority order.

Performance simulations indicate that our semantic locking technique maintains temporal consistency better than several other object locking techniques [10]. In [9] we prove that our Object Manager's semantic locking technique can bound imprecision in the database and that it can support global correctness by showing that it can enforce a form of Epsilon Serializability [17] specialized for object-oriented databases.

RTSORAC Transaction Management

Our Open OODB Transaction Policy Manager (TPM) provides for real-time scheduling of transaction processes, and transaction timing constraint enforcement.

Real-Time Scheduling. The real-time transaction scheduling performed by the TPM is essentially a mapping of timing constraints expressed in RTSORAC transactions into real-time POSIX priorities for transaction processes. This mapping is designed so that the transaction process priorities realize Earliest-Deadline-First (EDF) scheduling. EDF scheduling has been shown to be effective in real-time databases [1], but implementing EDF scheduling using the capabilities specified by the POSIX interface is non-trivial. The problem is that optimal EDF scheduling requires infinite priorities (one for each possible deadline), while POSIX mandates a minimum of only 32 priorities[1]. Furthermore, POSIX mandates a form of First-In-First-Out (FIFO) scheduling for processes of the same priority[2]. FIFO scheduling can adversely affect EDF scheduling since a later deadline may execute before an earlier deadline within same priority. Our TPM is designed to minimize the violation of EDF transaction scheduling order while using the capabilities of POSIX. It does this in three steps:

1. Initial priority assignment is done by mapping the RTSORAC transaction deadline to a POSIX process priority using the probability distribution of deadlines in the application. This mapping uses the distribution to equalize the expected number of processes at any given priority. For example, if most transaction processes have deadlines in the 50ms to 100ms range, this range might be split into several POSIX priorities while a range of several seconds in length might map to a single priority because there is a lower probability of RTSORAC transactions within that particular larger range of deadlines.
2. The TPM uses POSIX primitives to shuffle transaction processes within a priority so that they are queued in deadline order, not FIFO order.
3. The TPM increases priorities of transaction processes as time progresses. That is, when time passage causes a transaction process to map to a new

[1]POSIX mandates a minimum of 32 priorities for standard compliance; implementations may, and often do, provide more priorities.

[2]There are two other POSIX policies: *round robin* which is FIFO with a time quantum, and *other*, which is non-standard.

(higher) priority because its deadline is nearer, the TPM increases that transaction process's priority.

Details and simulation results of this scheduling technique are presented in [18].

Transaction Timing Constraint Enforcement. In addition to earliest deadline first scheduling on the processor, the TPM is also responsible for mapping RTSORAC transaction timing constraints to POSIX primitives for enforcing timing constraints. In particular, the TPM maps expressed earliest start times, deadlines, and periods into appropriate POSIX timer primitives. A RTSORAC transaction's earliest start time e is implemented by setting a timer for e and suspending the transaction process until the timer signal arrives. A RTSORAC transaction deadline d is implemented by setting a timer for d. If the timer signal arrives, it causes the transaction process to jump to the signal handler, which contains high-level RTSORAC enforcement rule (exception handling) code. Periodic execution requires repeatedly setting timers for the start and end of period frames. This enforcement procedure is described in [22].

4 CONCLUSION

This chapter has presented the RTSORAC model and its use in designing real-time extensions to the Open OODB system. The model supports expression of logical consistency, temporal consistency, and imprecision constraints as well as their trade-offs for both data objects and transactions. It also supports expression of complex data types and associations among data items. The prototype uses main-memory objects with semantic real-time concurrency control to achieve fast access that observes the semantics of the logical, temporal, and imprecision constraints.

We believe that real-time object-oriented database systems can be effective for many applications that involve management of complex, real-time data. The RTSORAC model is a useful abstraction of the incorporation of real-time requirements into object-oriented database systems. The prototyping of the model in the Open OODB system is an important step towards indicating the feasibility of the RTSORAC approach.

REFERENCES

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

[2] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *Proceedings of ACM SIGMOD Conference*, May 1994.

[3] A.P. Buchmann, D.R. McCarthy, M. Hsu, and U.Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *The Fifth International Conference on Data Engineering*, February 1989.

[4] Grady Booch. *Object-Oriented Design*. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

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

[6] MichaelJ. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley Publishing Company, 1989.

[7] Oscar Diaz and Peter M. D. Gray. Semantic-rich user-defined relationship as a main constructor in object-oriented databases. In R.A. Meersman, W. Dent, and S. Khosla, editors, *Object-Oriented Databases: Analysis,Design & Construction (DS4)*, pages 207 – 224. Elsevier Science Publishers, B.V. (North-Holland), 1991.

[8] 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.

[9] Lisa Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control with bounded imprecision. To appear *IEEE Transactions on Knowledge and Data Engineering*.

[10] Lisa Cingiser DiPippo and Victor Fay Wolfe. Performance of object-based semantic real-time concurrency control Submitted to *SIGMOD 97*.

[11] David Druin. ??? Master's Thesis. Dept. of Computer Science, The University of Rhode Island, 1996.

[12] J. Rumbaugh et. al. *Object-Oriented Modelling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[13] P. Fortier, JJ Prichard, and Victor Fay Wolfe. SQL/RT: Real-time database extensions to the SQL standard. To appear in *Standards and Interface Journal*, 1994.

[14] Paul Fortier, Victor Fay Wolfe, and JJ Prichard. Flexible real-time SQL transactions. In *IEEE Real-Time Systems Symposium*, Dec. 1994.

[15] IEEE. *Portable Operating System Interface (POSIX); Part 1: System API; Ammendment 1: Real-time Extension.* IEEE, 1994.

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

[17] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. to appear in *Transactions on Knowledge and Data Engineering.*

[18] Joseph Senerchia. A dynamic real-time scheduler for posix 1003.4a compliant operating systems. Master's Thesis. Dept. of Computer Science, The University of Rhode Island, 1993.

[19] John Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2, September 1990.

[20] Bhavani Thurasingham and Alice Schafer. RT-OMT: A real-time object modeling technique for designing real-time database applications. In *Proceedings of the Second IEEE Workshop on Real-Time Applications*, pages 124–129, July 1994.

[21] 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.

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

[23] 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.

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