Schema Evolution for Real-Time Object-Oriented Databases[1]

Lei Zhou, Elke A. Rundensteiner, and Kang G. Shin

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48105-2122
{lzhou, rundenst, kgshin}@eecs.umich.edu
Fax: (313) 763-4617

Technical Report CSE-TR-199-94, March 1994

It has been shown that the database schemata often experience considerable changes during the development and initial use phases of database systems for advanced applications, such as automated manufacturing and computer-aided design. An automated schema evolution system can significantly reduce the amount of work and potential errors related to schema changes. Although schema evolution for non-real-time databases was the subject of some previous research, its impact on real-time database systems remains unexplored. Since these advanced applications typically utilize object-oriented models to handle complex data types and there exists no agreed-upon real-time object model that can be used as foundation to define a schema evolution framework, we first design a conceptual real-time object-oriented data model, called ROMPP. It captures the key characteristics of real-time applications, namely, timing constraints and performance polymorphism. It uses specialization dimensions to model timing specifications and letter class hierarchies to capture performance polymorphism. We then re-evaluate previous (non-real-time) schema evolution support in the context of real-time databases, which results in several modifications to the semantics of schema changes and to the needs of schema change resolution rules and schema invariants. Furthermore, we expand the schema change framework with new constructs—including new schema change operators, new resolution rules, and new invariants—for handling additional features of the real-time object model.

Key words: Object-oriented database, real-time data model, performance polymorphism, letter class hierarchy, schema evolution.

1 INTRODUCTION

The object-oriented approach has been shown to be an effective way to manage the development and maintenance of large complex systems, including real-time systems [5, 7]. Many advanced real-time applications, such as manufacturing workstations/cells and air traffic control systems, require a built-in database management system (DBMS) to support concurrent data access and provide well-defined interfaces between different software entities (tasks, processes, and modules). They typically are subject to a range of timing constraints, which require the DBMS to provide timing guarantees, sometimes, under complex conditions. The deadlines of real-time tasks can be classified as *hard*, *firm*, or *soft* [26]. A deadline is said to be hard if the consequences of not meeting it can be catastrophic, such as in a nuclear reactor controller. A deadline is firm if the results produced by the corresponding task cease to be useful as soon as the deadline expires, but consequences of not meeting the deadline are not catastrophic, e.g., weather forecast. A deadline which is neither hard nor firm is said to be soft. The utility of results produced by a task with a soft deadline decreases over time after the deadline expires. Conventional DBMSs generally have no mechanisms to specify, and much less to enforce, such complex timing guarantees.

1. This research was supported in part by the Horace H. Rackham School of Graduate Studies at the University of Michigan under a Research Partnership Grant, the United Parcel Service Foundation under an IVHS Graduate Fellowship, and the National Science Foundation under Grant DDM-9313222.

Thus, they do not offer the performance levels or response-time guarantees needed by these real-time applications. Such inadequacy has recently spawned the field of real-time databases [11, 23, 26, 27, 30].

The requirements of a real-time system, like most other systems, are likely to change during its life cycle. The system must be able to evolve smoothly in order to improve its performance or to introduce new functionality, without disrupting existing services. The extent of changes in a typical working relational database system is illustrated in [28], which documents the measurement of schema evolution in the development and initial use of a health management system used in several hospitals. There was an increase of 139% in the number of relations and 274% in the number of attributes in the system during the nineteen-month period of study. In [18], significant changes (about 59% of attributes on the average) were reported for seven applications. These applications varied from project tracking, real estate inventory and accounting and sales management, to government administration of the skill trades and apprenticeship programs. It was observed that the most frequent contributor to schema changes is changing user requirements. Advanced database applications, especially engineering design applications, which are most appropriately captured by object-oriented databases (OODBs), are typically much less understood and thus are even more prone to numerous changes in the database schemata. In this paper, we investigate the impact of schema evolution on real-time applications. To the best of our knowledge, this question of real-time schema evolution has not been addressed before.

Real-time database research often uses the object-oriented paradigm. However, no agreed-upon real-time object-oriented data model is available at this time. Therefore, we first need to define a real-time data model, based on which we can develop a schema evolution framework. We have evaluated existing models used for real-time applications [3, 9, 12, 14, 16, 20, 31]. Based on this evaluation, we extract a simple yet powerful real-time object model that explicitly captures important characteristics of RTDB applications, especially in the manufacturing application domain, namely, timing constraints and performance polymorphism. It uses specialization dimensions to model timing specifications and letter class hierarchies to capture performance polymorphism.

We then develop a framework for changes to schemata of real-time OODBs based on the typical schema change taxonomy [2]. Schema evolution has been defined for simple (non-real-time) object-oriented models [2, 22, 24, 32]. We now re-evaluate this work in the context of real-time databases, which, as we will show, results in several modifications to the semantics of schema changes and to the needs of schema change resolution rules and schema invariants. Furthermore, we expand the schema change framework with new constructs—including new schema change operators, new resolution rules, and new invariants—for handling additional features of the real-time object model. We also demonstrate the utility of our real-time object model and schema evolution framework based on several manufacturing applications.

The remainder of the paper is organized as follows. Section 2 describes a conceptual real-time object model, while Section 3 defines a schema evolution framework based on the model. Section 4 briefly covers related work. We discuss additional research issues in Section 5, and conclude the paper with Section 6.

2 CONCEPTUAL REAL-TIME OBJECT MODEL

In this section, we describe a conceptual real-time object model, called ROMPP (*Real-time Object Model with Performance Polymorphism*). It is *conceptual* in the sense that it is not dependent on any specific implementation. This model aims to provide a simple, yet sufficiently powerful foundation, for our real-time schema evolution research by explicitly capturing the key characteristics of real-time applications.

2.1 Object-Oriented Concepts

ROMPP is object-oriented, that is, any real-world entity is represented by one modeling concept: an *object*. ROMPP adopts basic object-oriented concepts, such as class and inheritance, as can be found in most object-oriented models [6, 10, 15]. These concepts are defined below as needed for the remainder of this paper.

Definition 1. *An* **object** *is a triple (***identifier**, **state**, **behavior***), where the identifier is generated by the system and uniquely identifies the object, the state is determined by the set of values of the* **instance variables** *associated with the object, and the behavior corresponds to the* **methods** *associated with the object. An instance variable of an object can hold as value either a system-provided object, such as an integer, or a user-defined object, such as a Sensor. Instance variables are* **private** *to the object, i.e., they can only be accessed by the object's methods. A method is defined by (***signature**, **body***), where the signature consists of a method name M and a mapping from input parameter specifications to an output parameter specification:* $\mathrm{M}\,(\mathrm{In}_1, \mathrm{In}_2, \ldots, \mathrm{In}_n) \rightarrow \mathrm{Out}$. *A parameter specification (either input or output) is a class name. The body corresponds to the actual code which implements the desired functionality of the method. Methods can be either private or* **public***. A public method is accessible to all methods of the object or even to other objects. An instance variable V_i of an object A can be specified as being* **composite***. In this case, the object B referenced through the composite instance variable V_i is owned by the object A. Deletion of A will cause the deletion of B.*

Definition 2. *A* **class** *is a tuple (***name**, **structure***) that represents a group of objects with the same declaration of instance variables and methods. The name of a class is a string and the structure consists of the declaration of common instance variables and methods.*

Definition 3. *For two classes C_1 and C_2, C_1 is a* **subclass**[2] *of C_2, denoted C_1* **is-a** *C_2, if and only if C_1 inherits every instance variable and method of C_2.*

Multiple inheritance is allowed, that is, a class can have more than one superclass. Note that private instance variables and methods of a class are not visible to its subclasses, although they are inherited by the subclasses. Only public methods of the superclasses are accessible to the subclass and become part of its public interface. In other words, private instance variables inherited from a superclass are stored in the instances of the subclass, but these private instance variables (and methods) can only be accessed by the subclass via public methods defined in the superclass. A public method of a class can be declared virtual, i.e., it has no code associated with it and must be implemented in the class' subclasses (or descendants). The objects of the same class type are usually called *instances* of the class.

Definition 4. *A* **class hierarchy** *is a directed acyclic graph*[3] *$S = (V, E)$, where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on $V \times V$ that represents all subclass relationships between all pairs of classes in V. In particular, each directed edge e from C_1 to C_2, denoted by $e = \langle C_1, C_2 \rangle$, represents the is-a relationship (C_1 is-a C_2).*

2.2 Key Characteristics

Based on our evaluation of existing real-time systems [3, 9, 12, 14, 16, 20, 31] and real-time manufacturing applications [1, 4, 19], we have identified two key characteristics for real-time data models: *timing constraints* and *performance polymorphism.*

2.2.1 Timing Constraints

The first key characteristic is the concept of timing constraints. A real-time system must have the ability for the users to specify timing constraints and for the system to provide timing guarantees. Any real-time object model must thus have constructs to specify timing constraints. The implementation of a real-time DBMS must provide mechanisms to guarantee these deadlines.

Definition 5. *The* **timing constraint** *of a task refers to the deadline by which the task must be completed.*

In our real-time object model, timing constraints are associated with the performance of methods, since the behavior of an object is represented by its methods and applications will be requesting services from

2. Throughout this paper, we say that A is a *subclass* of B (B is a *superclass* of A) iff A inherits directly from B, and A is a *descendant* of B (B is an *ancestor* of A) iff A inherits directly or indirectly from B.
3. A class hierarchy without multiple inheritance corresponds to a tree rather than a DAG.

objects via their respective methods. We thus need to extend the definition of a method (Definition 1).

Definition 6. *A* **method** *in ROMPP is defined as a triple (***signature**, **body**, **performance***), with signature and body defined as in Definition 1. The (optional) third field specifies the performance measure of the method, such as execution time, memory space, etc.*

We shall see that the exact specification of the performance field of a method triplet differ depending on the type of classes, as described in the next subsection.

2.2.2 Performance Polymorphism

To implement the functionality of a method, typically several different algorithms and/or data structures can be used. Real-time systems need support in selecting one from these implementations based on, for instance, the performance constraints, since this would allow them to build complex real-time applications effectively. For example, a method that sorts n items may choose among a variety of sorting algorithms, such as insertion sort, merge sort, quick sort, counting sort and bucket sort, depending on the size of inputs and/or knowledge about the keys to be sorted on. A real-time application may want to select among these different sorting implementations based on performance characteristics, but without having to deal with details of the respective implementation. This second key characteristic of a real-time model is called performance polymorphism.

Definition 7. **Performance polymorphism** *refers to the concept of maintaining and selecting among multiple implementations of a method (body) that carry out the same task and differ only in their performance measures, such as execution time, memory space, system configuration, result precision, and so on. Performance polymorphism is explicitly supported by ROMPP, allowing dynamic selection of the most appropriate method implementation based on performance characteristics desired by the application.*

If a real-time object model does not have explicit constructs for performance polymorphism, we have to use one of the following approaches:

1. The knowledge of performance polymorphism is captured and maintained separately from the schema. For example, the service designer[4] may use a library to group different implementations of the same service. The knowledge about such real-time object libraries is not part of the system schema. Therefore, it is the application develop's responsibility to keep track of different implementations and, more importantly, about their relative characteristics and performance metrics. The application developer must use them appropriately in the improvement of existing systems or the development of new applications. Obviously, such approaches do not provide support for software reusability, and put all burden on the application developer.

2. The service designer could use one implementation of an object to meet all performance requirements, no matter how different they are. This over-simplistic approach would typically require us to assume a worst-case scenario. This is not even always possible, because requirements may contradict one another. It also wastes resources and poses true limitation on applications. For example, suppose the system has a memory space of 10MB, and the chosen implementation of object A requires 8MB while object B needs 3MB. Obviously, A and B cannot co-exist in memory. Therefore, a real-time task cannot receive services from A and B concurrently, even if A needs only 2MB to provide the desired services for this particular application when using a slightly slower algorithm.

3. Another option is to duplicate the definition of the method (or object) with each of its implementations and give them distinct names in order to simulate performance polymorphism. This would again carry all disadvantages of the first approach above, making the application developer responsible for maintaining information about individual services and their relationships. In addition, a system of such a type is difficult to maintain. Any change in the definition of the method has to be made to all its

4. In this paper, we distinguish between the *service designer* who builds the kernel classes required by an application, and the *application developer* who utilizes these kernel classes stored in the DBMS to construct applications.

duplicates, which is inefficient and often prone to errors.

Our model overcomes all of these problems by adopting the following strategies:

1. It provides a definition of the service offered by a method, and supports explicit association of distinct implementations with each service;
2. It allows for the explicit annotation of the performance features that characterize each implementation by the service designer, and for their explicit maintenance by the database system;
3. It supports an automatic mechanism for the application developer to work with the most appropriate implementation of a desired service based on requested performance requirements, without having to explicitly choose one of the implementations. Should the performance requirements of an application change, the mechanism would transparently rebind the requested service with the most appropriate implementation.

Performance polymorphism in ROMPP is captured by the *letter class hierarchy*, which is based on an object-oriented programming technique—the *envelope/letter structure* [8].

Definition 8. *An* **envelope/letter structure** *is a composite object structure formed by a pair of classes that act as one: an outer class (***envelope class***, or* **EC***) that is the visible part to the user, and an inner class (***letter class***, or* **LC***) that buries implementation details.*

Definition 9. *A* **letter class hierarchy** *is a class hierarchy as defined in Definition 4 that consists of, as its root, an envelope class and zero to many letter classes. The envelope class and all its letter classes must have exactly the same public methods. Furthermore, the letter classes can only have is-a relationships with classes in the same letter class hierarchy. Letter classes are not explicitly accessed by the application developer, but rather are manipulated by the system based on the performance requirements specified with the envelope class.*

In other words, the letter classes of a letter class hierarchy are all descendants of their corresponding envelope class. They can have is-a relationships between themselves, thus inheriting additional instance variables and methods. But they cannot have is-a relationships with any other envelope or letter classes.

Definition 10. *An* **envelope class hierarchy** *is a class hierarchy that consists of, as its root, a system-provided class, called* **ROOT***, and one to many envelope classes.*

Notice that the definition of an envelope class hierarchy does not include letter classes, although each envelope class has an associated letter class hierarchy. This emphasizes the fact that, for applications, letter classes are hidden behind their corresponding envelope classes. A public method of an envelope class can be designated as a *specialization dimension*, as defined below.

Definition 11. *A* **specialization dimension** *is a performance measure (Definition 6) that distinguishes letter classes from one another. A specialization dimension must be assigned to a public method in the letter class hierarchy. There is a* **specialization space** *associated with each letter class hierarchy and its axes are specialization dimensions.*

The letter classes specialize along one or more specialization dimensions that have been specified for the public methods in their corresponding envelope class. The most common specialization dimension for real-time applications is the execution time of a method. The public methods corresponding to a specialization dimension must be declared virtual in the envelope class. That is, there is no code attached to the methods with envelope classes. A public method could represent more than one specialization dimension. For example, if the implementation of a method requires a trade-off between execution time and memory space consumed, different implementations of the method will represent different points in a two-dimensional specialization space, whose axes are execution time and memory space consumed.

The performance-related information of a letter class hierarchy is reflected in its specialization space. A simple implementation of a specialization space would be to organize all letter classes in a letter class hierarchy into an unsorted linked list. A sequential search through the list would find the best letter class (if one exists) satisfying the given performance requirements. This simple approach would work well when the number of letter classes is small. For more efficient lookup, letter classes may be sorted along their specialization dimensions. Envelope classes have complete knowledge of how their corresponding letter class hierarchies are organized. This knowledge may be implicit when all letter class hierarchies use the same organization technique and it is known to the system, or explicit when the knowledge of the organization technique is stored in individual envelope classes. The relative performance of a letter class is significant in terms of its location in this specialization space. Hence any change on the performance value may map the letter class to a different point in its specialization space. Letter classes are not necessarily static (or predefined); they can be created at run-time.

2.3 Model Constructs

For the specification of the constructs introduced above, we propose the following data definition notation. Note that these model constructs are designed to be programming language independent. They are specified by statements with special key words preceded by the character "@". The following constructs have been defined:

1. @EC <ec>

   It declares that <ec> is an envelope class. This statement is used when defining classes.

2. @LC <lc> OF <ec>

   It declares that <lc> is a letter class of the envelope class <ec>, again used for class definition.

3. @DIM: <method> = <identifier>

   It specifies that <method> is a specialization dimension of the letter class hierarchy and gives it a unique identifier. This construct can only be used within the definition of an envelope class.

4. @DIM: <identifier> = { <value> | <expr> | unknown }

   It specifies the performance value of the specialization dimension <identifier> that has been declared for its corresponding envelope class. This construct can only be used in the context of letter classes.

Several examples are given in Section 2.4 to explain the newly introduced concepts. These examples are described in C++, since C++ and C are among the most popular programming languages for real-time applications. By placing the model constructs in programming language comments, we avoid modifying the programming language itself. The model constructs can be pre-processed, before the code is sent to the programming language compiler.

2.4 Examples

Example 1: A Letter Class Hierarchy with One Specialization Dimension

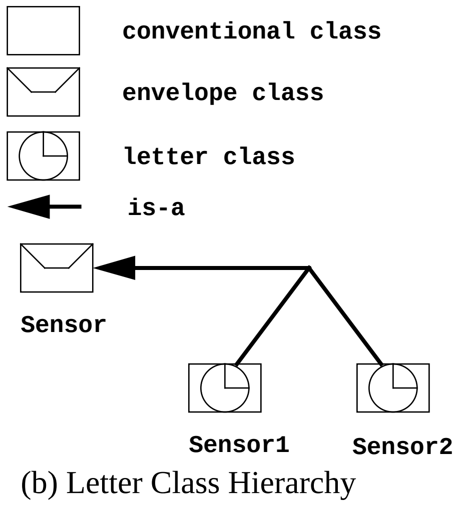
In Figure 1, the class **Sensor** is the envelope class, while classes **Sensor1** and **Sensor2** are its letter classes. The latter encapsulate two different implementations of the method **sample()** defined for the former. The method **sample()** has one associated specialization dimension, identified by **STime**. **STime** refers to requirements on execution time of the method, and the two letter classes associate different values of execution time, i.e., timing guarantees, with **sample()**. In the example, **sample()** is the only specialization dimension. Therefore, the specialization space is one-dimensional as shown in Figure 1(c).

Example 2: A Letter Class Hierarchy with Two Specialization Dimensions

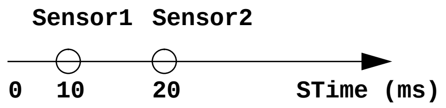
In the example depicted in Figure 2, there are two specialization dimensions, associated with the methods **sample()** and **process()**, respectively. Therefore, the specialization space is a plane, as shown in Figure 2(c). Note that specialization dimensions may not necessarily be inferred from the structure of the letter class hierarchies as, for instance, shown in Figure 2(b), since these simply capture is-a relationships in terms of property inheritance.

```
// @EC: Sensor
class Sensor {
public:
    Sensor();
    // @DIM: int sample() = STime
    virtual int sample();
    ....
}

// @LC: Sensor1 OF Sensor
class Sensor1 : public Sensor {
public:
    Sensor1();
    // @DIM: STime = 10 ms
    int sample();
    ....
}

// @LC: Sensor2 OF Sensor
class Sensor2 : public Sensor {
public:
    Sensor2();
    // @DIM: STime = 20 ms
    int sample();
    ....
}
```

(a) Model Description



(b) Letter Class Hierarchy

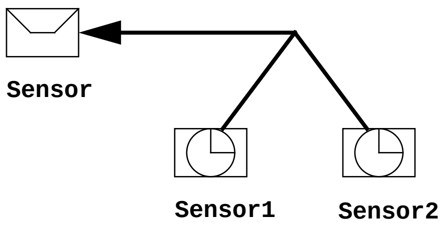


(c) Specialization Space

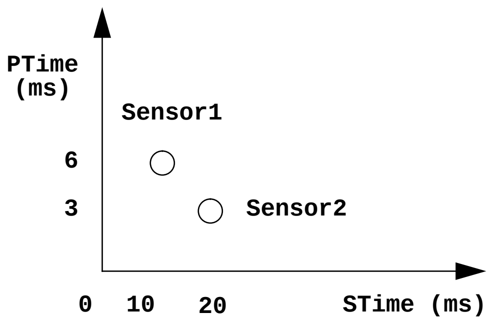
Figure 1. Example of One-dimensional Specialization Space

```
// @EC: Sensor
class Sensor {
public:
    Sensor();
    // @DIM: int sample() = STime
    virtual int sample();
    // @DIM: void process() = PTime
    virtual void process();
    ....
}
// @LC: Sensor1 OF Sensor
class Sensor1 : public Sensor {
public:
    Sensor1();
    // @DIM: STime = 10 ms
    int sample();
    // @DIM: PTime = 6 ms
    void process();
    ....
}
// @LC: Sensor2 OF Sensor
class Sensor2 : public Sensor {
public:
    Sensor2();
    // @DIM: STime = 20 ms
    int sample();
    // @DIM: PTime = 3 ms
    void process();
    ....
}
```

(a) Model Description



(b) Letter Class Hierarchy



(c) Specialization Space

Figure 2. Example of Two-dimensional Specialization Space

2.5 Real-Time Object-Oriented Database Schema

Definition 12. *A* **real-time object-oriented database (RTOODB) schema** *is composed of one envelope class hierarchy and a set of zero or more letter class hierarchies, defined in Definition 10 and Definition 9, respectively. Each letter class hierarchy is associated with one envelope class.*

If an envelope class has no letter classes, it degenerates to a conventional class. Therefore, a RTOODB schema is comprised of exactly one envelope class hierarchy and zero to many letter class hierarchies. The root of the envelope class hierarchy is the system provided class **ROOT**, while the root of a letter class hierarchy is its corresponding envelope class.

Example 3: A RTOODB Schema

Figure 3 shows an example RTOODB schema. The shaded area is an envelope class hierarchy, which is visible to the application. We now demonstrate how this schema can be used by an application developer. Suppose that the rightmost letter class hierarchy (enclosed in the rounded rectangle) is the same as that in Example 2 (Figure 2), i.e., a letter class hierarchy with a two-dimensional specialization space.

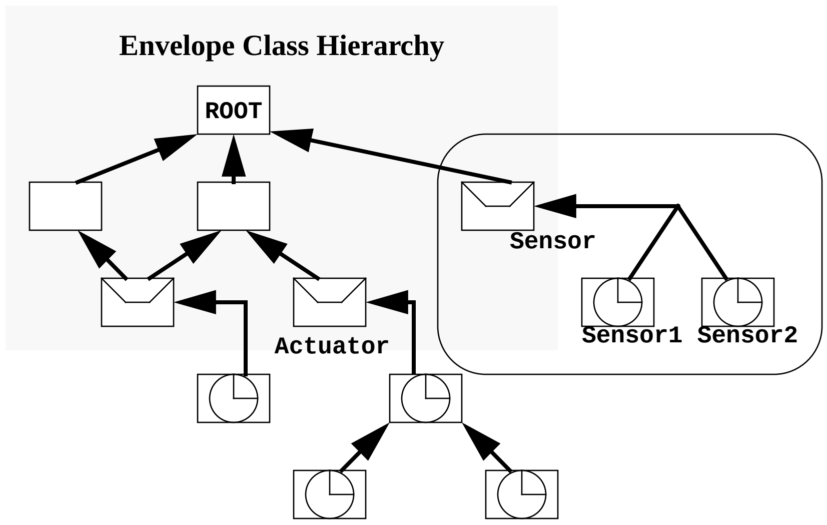


Figure 3. Example Real-Time Object-Oriented Database Schema

Assume that an application requires a **Sensor** object with the following constraints:

```
Class Foo {
public:
    ...
private:
    Sensor s(“STime<=15ms, PTime<7ms”);
    ...
}
```

Then an object of **Sensor1** will be constructed by our system since it satisfies constraints on both **STime** and **PTime**. If in the future, the application adjusts its requested timing requirements for the **Sensor** object to “**STime<22ms, PTime<5ms**”, then the system will automatically select another implementation object for **Sensor**, namely, an object instance of class **Sensor2**, replacing the initial choice of a **Sensor1** object. This process of rebinding will be *transparent* to the application developer, since our model supports true performance polymorphism.

3 REAL-TIME DATABASE SCHEMA EVOLUTION

The requirements of a real-time system, like most other systems, are likely to change during its life cycle. The system must be able to evolve smoothly in order to improve its performance or to introduce new functionality, without disrupting existing services. If the service designer adds a new implementation **Sensor3** to the schema in Figure 3, for instance, the execution of existing applications (e.g.,the class **Foo** in Example 3) should not be disrupted by this schema change. More important, our system may direct existing applications to use the newly added implementation, if it is more appropriate for the requested performance requirements, due to our support of performance polymorphism.

Having designed the real-time object model, we can now proceed with our task of defining schema evolution. For this purpose, we need to expand the typical steps of schema evolution [2] to the real-time object model:

1. Identify a schema change taxonomy. We need to determine which schema changes are meaningful, given the new definition of a RTOODB schema.
2. Identify schema change invariants. In order to keep the consistency of the schema across different modifications, these invariant properties must be preserved.
3. Design schema change rules. When there are alternative ways to do a schema change without violating any of the invariants, rules are designed to eliminate ambiguity.
4. Define schema change semantics. The effect of each schema change identified in step 1 on the rest of the schema is investigated and its impact on the underlying data is also considered.

3.1 Schema Change Taxonomy

One of the first object-oriented schema change approaches has been proposed by Banerjee *et al.* [2] for ORION.[5] Note that this taxonomy, adopted in most other schema evolution research for OODBs [22, 24, 32], still corresponds to the most frequently used set of schema changes. In fact, most commercial OODB systems have implemented a subset of this taxonomy as their schema change support [10, 13, 24]. None of these approaches considers real-time models. We adopt a similar schema change taxonomy, however, with extensions necessary for changes on real-time constructs of the schema. A complete description of our ROMPP schema change taxonomy is given below:

(1) Changes to the contents of a node (a class)
(1.1) Changes to an instance variable
(1.1.1) Add a new instance variable to a class
(1.1.2) Drop an existing instance variable from a class
(1.1.3) Change the name of an instance variable of a class
(1.1.4) Change the inheritance (parent) of an instance variable
(1.1.5) Drop the composite property of an instance variable
(1.2) Changes to a method
(1.2.1) Add a new method to a class
(1.2.2) Drop an existing method from a class
(1.2.3) Change the name of a method of a class
(1.2.4) Change the body of a method in a class
(1.2.5) Change the inheritance (parent) of a method
(1.2.6) Make a method a new specialization dimension
(1.2.7) Drop the specialization dimension property of a method
(2) Changes to an edge
(2.1) Make a class S a superclass of a class C

5. ORION is an OODB system built at MCC in Austin, Texas. A commercial product version of ORION is being marketed as ITASCA [13].

(2.2) Remove a class S from the superclass list of a class C
(2.3) Change the order of superclasses of a class C
(3) Changes to a node
(3.1) Add a new class
(3.2) Drop an existing class
(3.3) Change the name of a class

Although a number of schema changes in our taxonomy are the same as those in [2], we show in Section 3.4 that the semantics of these changes are quite different. In order to support changes of our model, we now must evaluate changes on both letter class and envelope class hierarchies. Furthermore, there are two additional schema changes, "(1.2.6) Make a method a new specialization dimension" and "(1.2.7) Drop the specialization dimension property of a method", which are unique to real-time models.

3.2 Schema Change Invariants

In order for any schema change to be meaningful, i.e., to maintain a correct database, it must guarantee the consistency of the schema. We thus need schema invariants to define the correctness of schema properties. We have adopted the invariants proposed in [2] with some modifications:

1. *Class Hierarchy Invariant*. The class hierarchy is a rooted and connected directed acyclic graph with uniquely named nodes (classes) and unlabeled edges (subclass relationships) (see Definition 4).
2. *Distinct Name (Signature) Invariant*. All instance variables of a class must have distinct names. Similarly, all methods of a class must have distinct signatures (see Definition 6).
3. *Distinct Origin Invariant*. All methods of a class have distinct origins.[6]
4. *Full Inheritance Invariant*. A class inherits all instance variables and methods from each of its superclasses, except when full inheritance causes a violation of the distinct name (signature) and distinct origin invariants. Only public methods are visible to the class and its descendants.

Moreover, we address the consistency requirements specific to our real-time object model by introducing the following additional invariants.

5. *Envelope Class Hierarchy Invariant*. There is only one envelope class hierarchy in the schema and it must satisfy the Class Hierarchy Invariant.
6. *Letter Class Hierarchy Invariant*. There may be zero or more letter class hierarchies and each of them must satisfy the Class Hierarchy Invariant.
7. *Envelope/Letter Class Relationship Invariant*. The declaration of any public method of a letter class must match that of its corresponding envelope class, and *vice versa*.
8. *Specialization Dimension Invariant*. Each specialization dimension has a unique identifier, which is specified for a method in an envelope class. The identifier can only be referenced with the same method of the letter classes associated with the envelope class.

3.3 Schema Change Rules

We adopt rules similar to those defined in [2] and enhance them for our real-time object model. They apply to both envelope class hierarchies and letter class hierarchies.

1. If a method is defined within a class C, and its declaration is the same as that of a method of one of its superclasses, the locally-defined method is selected over that of the superclass.
2. If two or more superclasses of a class C have methods with the same declaration but distinct origin, the method selected for inheritance is that from the first superclass among conflicting superclasses.

6. Since instance variables are private and invisible to subclasses, they always have distinct origins.

3. If two or more superclasses of a class C have methods with the same origin, the method of the first superclass is inherited by C.
4. When a method in a class C is changed, the change is propagated to all descendants of C that inherit the method, unless it has been re-defined within the descendants.
5. If a newly added public method, or a signature change to a public method, encounters any signature conflicts in the class or its descendants as a consequence of this schema modification, this change is rejected. For the purposes of propagation of changes to descendants, Rule 5 overrides Rule 2.
6. If a class A is made a superclass of a class B, then A becomes the last superclass of B. Thus, any method signature conflicts, which may be triggered by the addition of this superclass, can be ignored.
7. If class A is the only superclass of class B, and A is removed from the superclass list of B, then B is made an immediate subclass of each of A's superclasses. The ordering of these new superclasses of B is the same as the ordering of superclasses of A. A corollary to this rule is that, if the class **ROOT** is the only superclass of a class B, any attempt to remove the edge from **ROOT** to B is rejected.
8. If no superclasses are specified for a newly added envelope class, the class **ROOT** is the default superclass. A superclass must be specified for a newly added letter class.
9. For the deletion of edges from A to its subclasses, Rule 7 is applied if any of the edges is the only edge to a subclass of A. Further, any attempt to delete a system-defined class, e.g., **ROOT**, is rejected.
10. The composite property may be dropped from a composite instance variable; however, it may not be added to a non-composite instance variable.
11. If a composite instance variable of an object X is changed to non-composite, X disowns object Y which it references through the instance variable. The object X continues to reference the object Y; however, deletion of X will not cause Y to also be deleted.

In addition, we identify the following real-time-specific rules:

12. Letter classes are dependent on their corresponding envelope classes. That is, deletion of an envelope class will cause the deletion of its letter classes, and letter classes cannot exist before their corresponding envelope classes exist. This rule is based on the semantics of the letter class hierarchy concept given in Definition 9.
13. Changes to an envelope class must be propagated to its letter classes. This is to maintain the consistency of the letter class hierarchy (Definition 9) and the Full Inheritance Invariant.
14. The public interface of letter classes may not be changed, unless the changes are initiated by their corresponding envelope classes and propagated to letter classes. That is, no direct addition or alteration of the declarations of letter classes' public methods is allowed.

3.4 Schema Change Semantics

All changes to a RTOODB schema can be classified into the following two categories:

1. Changes to the envelope class hierarchy
2. Changes to letter class hierarchies

In general, changes to the envelope class hierarchy affect its corresponding letter class hierarchies, while changes to letter class hierarchies have no impact on the envelope class hierarchy. We define the semantics of these two categories of schema changes in the following two subsections.

It is often dependent on individual applications whether it is meaningful to convert existing instances of a class to that of the modified class. In real-time systems, for example, some objects have only a very short life-time; thus, it may not be necessary to keep them around after a certain period of time. Therefore, we only describe the impact of schema changes on existing data without worrying about if and when they are actually converted. One approach of converting existing instances of the affected letter class to instances of the new letter class is to provide forward mapping functions or update methods [21].

3.4.1 Schema Changes to a Letter Class Hierarchy

(1) Changes to the contents of a node (a class)
(1.1) Changes to an instance variable
(1.1.1) Add a new instance variable to a class

The descendants of the class are informed of the change in order to adjust their memory allocation. This schema change is almost always accompanied by other changes, e.g., ones that modify methods to use the new instance variable. Adding new instance variables seldom by itself affects the behavior of the class and its descendants. But in some cases, it could have an impact. For example, when the new instance variable demands significant amount of memory space, it can affect the performance of some methods. If it does, the letter class hierarchy specialization space may need to be reorganized.[7] This change affects existing instances of the class.

(1.1.2) Drop an existing instance variable from a class

The descendants of the class are informed of the change. This may cause consistency problems, since some methods may still be using the dropped instance variable. Therefore, such a change is usually accompanied by other changes, e.g., ones that modify the methods using the instance variable. The descendants of the class are informed of the change, in order to adjust their memory allocation. This change seldom by itself affects the behavior of the class and its descendants. If it does, the specialization space may need to be reorganized. This change affects existing instances of the class.

(1.1.3) Change the name of an instance variable of a class

No specialization space reorganization is needed. All methods using the instance variable need to be updated to utilize the new name. In general, existing instances of the affected letter classes may be used directly as the instances of corresponding new letter classes. No instance conversion is needed.

(1.1.4) Change the inheritance (parent) of an instance variable

It could have the same impact on method performance as in (1.1.1). Since instance variables are private and not visible to subclasses, this change can only be the side effect of schema changes (2.2) and (2.3). This change affects existing instances of the class.

(1.1.5) Drop the composite property of an instance variable

Rules 11 and 12 apply. A composite instance variable may be changed to non-composite, but not the opposite. This change is propagated to the descendants of the class. This change affects existing instances of the class.

(1.2) Changes to a method

For all changes to a method, existing instances of the affected letter classes can be used directly as the instances of the corresponding new letter classes, without requiring any conversion.

(1.2.1) Add a new method to a class

If the new method is public, the change is not allowed unless it is initiated by the corresponding envelope class (Rule 15). In this case, the change is made to the root of the letter class hierarchy and then propagated to all letter classes (Rule 14). Such a change may affect the specialization space, if the new method represents a new specialization dimension. If the change causes any conflicts, it is rejected (Rule 5). If the new method is private, the change is not visible to the descendants of the class. This change does not affect the specialization space.

(1.2.2) Drop an existing method from a class[8]

If the method is public, the change is not allowed unless it is initiated by the corresponding envelope class. In this case, it must be propagated to all letter classes. Such a change may affect the specialization space, because the dropped method may have represented a specialization dimension or it may have

7. If the specialization space is organized as a linked list, as mentioned in Section 2.2, no reorganization will be needed. If it is organized as an ordered list, then it will have to be re-inserted in the correct position of the list, depending on the new performance value.

8. The impact of schema changes on behaviors of objects is referred to as the behavior consistency problem in [32].

overridden some method that would now cause a performance change for other methods that use it. If the method is private, the change is not visible to the descendants of the class. All other methods using the dropped method need to be updated using additional schema changes.

(1.2.3) Change the name of a method of a class

If the method is public, the change is not allowed unless it is initiated by the corresponding envelope class. In this case, it must be propagated to all letter classes. If the method is private, the change is not visible to the descendants of the class. It does not affect the specialization space and existing instances.

(1.2.4) Change the body of a method in a class

If the method is public, the change must be propagated to the descendants of the class (Full Inheritance Invariant). The performance of the method needs to be re-evaluated, in order to determine a new performance value for each associated dimension.[9] If the method is private, the change is not visible to the descendants of the class. Such a change may affect the specialization space, as demonstrated by the example in Section 3.4.3. Providing code to a previously empty method body is a special case of this change.

(1.2.5) Change the inheritance (parent) of a method

The current method is dropped and the one from the new parent is added. If the method is public, the change must be propagated to all descendants (Rule 13), or rejected if it encounters any conflicts (Rule 5). Such a change may affect the specialization space.

(1.2.6) Make a method a new specialization dimension

The change is not allowed unless it is initiated by the corresponding envelope class. In this case, it must be propagated to all letter classes. The specialization space has one more dimension now and may need to be reorganized.

(1.2.7) Drop the specialization dimension property of a method

The change is not allowed unless it is initiated by the corresponding envelope class. In this case, it must be propagated to all letter classes. The specialization space has one fewer dimension now and may need to be reorganized.

(2) Changes to an edge

(2.1) Make a class S a superclass of a class C

C must be a letter class and S must be in the letter class hierarchy associated with C. S is made the last one in C’s superclass list. C now inherits public methods from S. Any signature conflicts may be ignored since S is the last of C’s superclasses. It may change C’s position in the specialization space and the space may need to be reorganized. This change affects existing instances of the class C.

(2.2) Remove a class S from the superclass list of a class C

C removes its methods inherited from S. Some methods from C’s other superclasses may become visible now. If S is the only superclass of C, S must not be an envelope class (Definition 9). In this case, let S’s superclass(es) be C’s superclass(es), in the same order. It may change C’s position in the specialization space and the space may need to be reorganized. This change needs to be propagated to C’s descendants. This change affects existing instances of the class C.

(2.3) Change the order of superclasses of a class C

This has no effect, if there are no method signature conflicts; otherwise, use Rules 2 and 3. For example, if a method M is defined in both superclasses S_1 and S_2, and S_2 is now before S_1 in C’s superclass list, the method M defined in S_2 is inherited instead that in S_1. This change affects existing instances of the class C.

(3) Changes to a node

(3.1) Add a new class

An envelope class must be specified as its parent (Rule 8); otherwise, the change is rejected. It adds a new point in the specialization space. The new class has no instances.

(3.2) Drop an existing class

9. The performance can be either analyzed and determined empirically by the service designer or evaluated by an automated analysis system. Details of this topic are beyond the scope of this paper.

The class to be dropped must be a letter class, i.e., it cannot be the root of the letter class hierarchy. If the class has any children, perform (2.2) for each of them. It removes a point in the specialization space. The user may choose to either drop its existing instances or convert them to instances belonging to its superclass(es).

(3.3) Change the name of a class

It brings no change to the specialization space. This name change may need to be registered with the corresponding envelope class. It may require its subclasses to change their parent's name.

3.4.2 Schema Changes to an Envelope Class Hierarchy

In general, changes to an envelope class hierarchy have similar semantics to those defined in [2]. In addition, the changes must be propagated to the corresponding letter classes, if any, as defined above, since letter class hierarchies are dependents of their corresponding envelope classes. They may cause reorganizations of the specialization spaces associated with letter class hierarchies. Because an envelope class acts as an interface to the user while the letter classes encapsulate implementation details (Definition 8), an envelope class is not allowed to have any instances unless it is degenerate. In this case it has no letter classes. The following example demonstrates how schema change invariants and rules are used to define the semantics of changes to an envelope class hierarchy.

(3.2) Drop an existing class

Drop it and its associated letter class hierarchy (Rule 12). If the envelope class has any subclasses (envelope classes, but not letter classes), perform (2.2) for each of them (Full Inheritance Invariant). Existing instances of its letter classes are dropped. The envelope class itself has generally no instances, unless it is degenerate. In the latter case, its instances are also dropped.

The following two schema changes are unique to the real-time object model. Their semantics for an envelope class hierarchy are different from that for a letter class hierarchy. These changes can be made to an envelope class as needed, but such changes to a letter class are not allowed unless preceded by the same change to the corresponding envelope class.

(1.2.6) Make a method a new specialization dimension

The change must be propagated to all corresponding letter classes, and the performance value of the corresponding method is set to "unknown" in the letter classes (see the example in the next subsection). The specialization space has one more dimension now and may need to be reorganized.

(1.2.7) Drop the specialization dimension property of a method

The change must be propagated to all corresponding letter classes. The specialization space has one fewer dimension now and may need to be reorganized.

3.4.3 An Example of Schema Changes

Suppose we have the following letter class hierarchy (Figure 4), which is very similar to the example in Figure 2. Class **Sensor** is an envelope class, and **Sensor1** and **Sensor2** are two letter classes. There is one specialization dimension, **STime**, corresponding to the execution time of the method **sample()**.

The first schema change the service designer makes is to add a new specialization dimension, **PTime**, to the method **process()**: "**ADD DIM PTime TO void process() IN Sensor**". According to the semantics defined earlier (schema change (1.2.6) in Section 3.4.1), the change must be made to the envelope class and then propagated to all its letter classes (and all its envelope class descendants). The schema evolution system defines the new specialization dimension at line 6 in **Sensor**, which causes the addition of new performance measures associated with all occurrences of the **process()** method (line 16 and line 26). Since the system does not know the performance of the method **process()** in letter classes yet, it puts "**unknown**" there (Figure 5). Now the letter class hierarchy has a two-dimensional specialization space.

Assume that next the service designer changes the body of the method **process()** in class **Sensor1**, using the command "**MODIFY void Sensor1::process() BODY = { <code> }**". This schema change is also available in non-real-time object models, but it has different semantics in the real-time case. That is, after changing the code, an updated performance value must be provided since the method is associated with a specialization dimension. Suppose the worst-case execution time for this particular implementation of **process()** in **Sensor1** is 6 ms, the system modifies the performance measure associated with the method accordingly (see line 16 in Figure 6).

```
1   // @EC: Sensor
2   Class Sensor {
3   public:
4       // @DIM: int sample() = STime
5       virtual int sample();
6
7       virtual void process();
8       ...
9   }
10
11  // @LC: Sensor1 OF Sensor
12  Class Sensor1 : public Sensor {
13  public:
14      // @DIM: STime = 10 ms
15      int sample();
16
17      void process();
18      ...
19  }
20
21  // @LC: Sensor2 OF Sensor
22  Class Sensor2 : public Sensor {
23  public:
24      // @DIM: STime = 20 ms
25      int sample();
26
27      void process();
28      ...
29  }
```

Figure 4. An Example of Schema Changes

```
1   // @EC: Sensor
2   Class Sensor {
3   public:
4       // @DIM: int sample() = STime
5       virtual int sample();
6       // @DIM: void process() = PTime
7       virtual void process();
8       ...
9   }
10
11  // @LC: Sensor1 OF Sensor
12  Class Sensor1 : public Sensor {
13  public:
14      // @DIM: STime = 10 ms
15      int sample();
16      // @DIM: PTime = unknown
17      void process();
18      ...
19  }
20
21  // @LC: Sensor2 OF Sensor
22  Class Sensor2 : public Sensor {
23  public:
24      // @DIM: STime = 20 ms
25      int sample();
26      // @DIM: PTime = unknown
27      void process();
28      ...
29  }
```

Figure 5. After Adding a New Specialization Dimension

```
// @LC: Sensor1 OF Sensor
Class Sensor1 : public Sensor {
public:
    // @DIM: STime = 10 ms
    int sample();
    // @DIM: PTime = 6 ms
    void process();
    ...
}
```

Figure 6. After Changing the Code for **process()** in **Sensor1**

4 RELATED WORK

There has been considerable work on defining schema evolution for OODBs. Examples are schema evolution for ORION [2], O_2 [32], GemStone [24], and GOOSE [22]. However, to the best of our knowledge, there has not been any work on defining schema evolution in the context of real-time OODBs. We adopt the typical steps of schema evolution and expand them for the real-time object model (see Section 3).

While a large body of work on real-time systems exists, no agreed-upon, conceptual model for real-time databases has been established. In this paper, we show that timing constraints and performance polymorphism are two key characteristics for the real-time applications and should be explicitly supported by an real-time data model. CHAOS (Concurrent Hierarchical Adaptable Object System) [3, 25] is an object-based language and programming/execution system designed for dynamic real-time applications. One of its key components is a C-based run-time library for the real-time kernel. CHAOS supports a limited form of dynamic parameterization of generic classes to allow easy development of different implementations of objects. Objects can be adapted at run-time, such as switching in different versions of object methods, changing the degree of concurrency, or changing the relative priorities of object methods. The parameterization of generic classes in CHAOS can be directly modeled by ROMPP, where envelope classes can represent generic classes and letter classes correspond to different implementations. These letter classes are specialized along several dimensions—the parameterized attributes in CHAOS.

ARTS (Advanced Real-time Technology) [20, 29] is a distributed real-time operating system kernel. RTC++ [12] is an extension of C++. Both of them are based on the same real-time object model, which describes real-time properties in systems and encapsulates rigid timing constraints in an object. Each object is composed of data, one or more threads of execution, and a set of exported operations. In this model, there are active objects—objects having one or more threads that can be executing when a message arrives. If an active object is defined with timing constraints for its methods, it is called a real-time object. In this real-time object model, the schedulability of a task set is easily analyzed under the rate monotonic scheduling. Unfortunately, performance polymorphism is not directly supported by the model. The use of real-time object libraries is suggested by the authors to remedy this. As discussed in Section 2.2.2, this is an undesirable solution in comparison with direct support of performance polymorphism. In ROMPP, we address this issue by explicitly supporting performance polymorphism, using the letter class hierarchy concept.

Flex [14] is a derivative of C++. It supports two modes of flexible real-time programs, designed to adjust execution times so that all important deadlines are guaranteed to be met. First, it allows computations to return imprecise results. Programs can be carried out as iterative processes that produce more refined results as more time is permitted, or they can use the divide-and-conquer strategy that provides partial results along the way. Second, it supports multiple versions of a function that carry out a given computation. These versions all perform the same task and differ in the amount of time and resources they consume, the system configuration to which they are adapted, the precision of the results that they return, and other performance criteria. The letter class hierarchy of ROMPP capturing the performance polymorphism corresponds closely to the second feature of Flex. A letter class may also be implemented using the imprecise computation technique. In other words, the first technique of Flex is simply one of several possible approaches for guaranteeing the timing constraints of actual method implementations. In Flex, several language primitives are provided to describe the alternative implementations of a method in the class, their performance, and the goals, such that the system may make appropriate selections as needed. This approach is not as flexible as the letter class hierarchy. For example, with the letter class hierarchy, letter classes can have different additional private data and/or methods if needed. Also, the knowledge about the characteristics of the letter classes may be stored in individual envelope classes, such that different binding procedures may be chosen for different letter class hierarchies.

HiPAC (High Performance ACtive database System) [9] combines databases with rule capabilities. Rules in HiPAC are first-class objects. A rule, among other features, allows the specification of its timing and other properties. When instances of the same class of rules are applied to different situations or objects, they may have different timing specifications. HiPAC does not have performance polymorphism and it does not make extensive use of most object-oriented features like classes or inheritance. Obviously, the letter class hierarchy can be used to model this characteristic of rules, where an envelope class represents a generic rule (or a class of rules) and letter classes represent the same rule with different timing specifications, which may require different implementations.

Wolfe *et al.* [31] propose an object-oriented model that supports the explicit specification of timing constraints, however, performance polymorphism is again not provided.

MDARTS (Multiprocessor Database Architecture for Real-Time Systems) [16, 17], developed at the University of Michigan, supports explicit declarations of real-time requirements and semantic constraints within application code. It examines these declarations during application initialization and dynamically adjusts its data management strategy accordingly. The research reported in this paper is an integral part of the ongoing MDARTS project. Specifically, we have extracted a conceptual real-time object model ROMPP and investigated the impact of schema evolution on real-time data models.

5 DISCUSSIONS

We now want to demonstrate the utility of the real-time object model ROMPP defined in Section 2. In particular, we want to discuss how it can be used to build real-time applications. The key concepts of ROMPP, namely, using specialization dimensions to characterize timing constraints and using letter class hierarchies to capture performance polymorphism are incorporated in MDARTS [16, 17]. MDARTS is a multiprocessor database architecture for real-time systems, being built in C++ in the Unix environment at the University of Michigan. To evaluate the suitability of the MDARTS in the domain of real-time manufacturing control applications, a prototype motion controller for a six degree-of-freedom robotic manipulator was implemented (Figure 7). It is a physical mechanism for geometric error compensation at the assembly stage of automotive applications. This mechanism includes a multi-axis manipulating device (essentially a robotic table to which sheet metal parts can be affixed), a multi-axis servo-motion controller which handles the execution of desired motions at the manipulator joint level. The servo-motion controller board is a Programmable Multi-Axis Controller (PMAC) designed and manufactured by Delta Tau Systems. The manipulator consists of a fixed base, a movable platform, and six independently positioned legs. Each leg is connected to the base by a 2-DOF joint on one end, and to the platform by a 3-DOF joint on the other end. The tops of adjacent legs are joined together at the platform connection point, forming a set of three leg triangles.

Timing constraints for data access were specified in order to ensure that all computations can be finished within each (periodic) control cycle. Although multiple implementations of the same service were not necessary for this application, the mechanism of selecting the most appropriate one among available implementations was in place and functioning. It was shown that MDARTS is able to monitor and modify the path of the manipulator while it is executing a sequence of move commands. This experiment demonstrates, among other features of MDARTS, that our real-time object model is useful in practice.

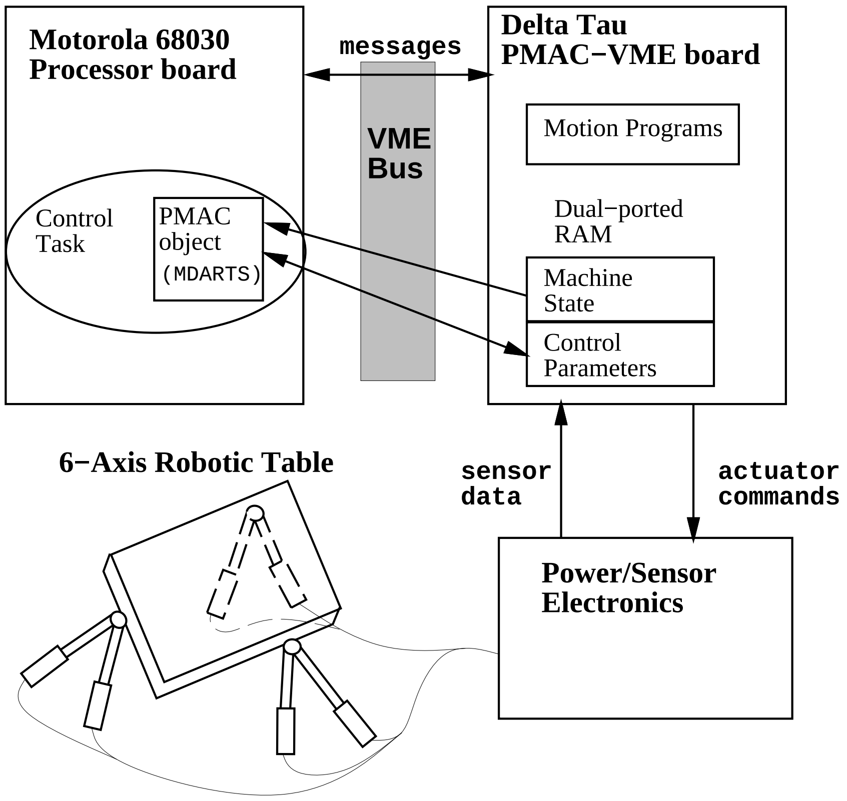


Figure 7. MDARTS Experiment Setup (reprinted with permission)

There are still several open questions to be answered. In particular, we are interested in investigating how to analyze the performance of methods, especially when they run concurrently, how to enhance the real-time object model by introducing more sophisticated constructs that allow, for instance, value propagation (e.g., propagation of the performance value of a method to other methods that use it) and conditional specifications (e.g., performance dependency on system configuration), and how to support on-line schema evolution in real-time.

6 CONCLUSIONS

In this paper, we proposed solutions to the as of now unaddressed area of schema evolution for real-time OODBs. Schema evolution support is becoming increasingly important, as advanced real-time applications, such as manufacturing systems, are starting to demand database services, rather than *ad hoc* data repositories, in order to reuse system components and to reduce the amount of work related to improving existing systems and developing new applications. Such applications must be flexible with revamping an existing system based on changes of technology and/or environment. They also need support to quickly configure new customized systems.

In the paper, we identified timing constraints and performance polymorphism as two key characteristics of real-time applications. We then presented a conceptual real-time object model, ROMPP, which explicitly captures these two features. We re-evaluated previous (non-real-time) schema evolution work in the context of real-time databases, which results in several modifications to the semantics of schema changes and to the needs of schema change resolution rules and schema invariants. Furthermore, we expanded the schema change framework with new constructs—including new schema change operators, new resolution rules, and new invariants—for handling features specific to the real-time aspects of ROMPP. There are still many open research questions to be answered, such as how to analyze the performance of methods that may run concurrently and how to support schema evolution in real-time. We believe our research is a good first step to explore the virgin area of schema evolution for real-time databases, and will cause new research efforts to spring up.

7 REFERENCES

[1] B. Anderson, “Next Generation Workstation/Machine Controller (NGC),” *Proc. IPC’92,* April 1992, pages xix-xxvi.

[2] Jay Banerjee, et al., “Semantics and Implementation of Schema Evolution in Object-Oriented Databases,” *SIGMOD 1987,* pages 311-322.

[3] Thomas E. Bihari, and Prabha Gopinath, “Object-Oriented Real-Time Systems: Concepts and Examples,” *IEEE Computers,* December 1992, pages 25-32.

[4] Sushil Birla, “A Conceptual Framework for Modeling Manufacturing Automation,” *Directed Study Report*, Department of Electrical Engineering and Computer Science, The University of Michigan, September 1993.

[5] Grady Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

[6] Paul Butterworth, Allen Otis, and Jacob Stein, “The Gemstone Object Database Management System,” *Communications of the ACM,* Vol. 34, No. 10, October 1991, pages 64-77.

[7] R.G.G. Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison-Wesley, 1991.

[8] James Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[9] U. Dayal, et al., “The HiPAC Project: Combining Active Databases and Timing Constraints,” *SIGMOD Record*, Vol. 17, No. 1, March 1988, pages 51-70.

[10] O. Deux, et al., “The O_2 System,” *Communications of the ACM*, Vol. 34, No. 10, October 1991, pages 34-48.

[11] Marc H. Graham, “Issues in Real-Time Data Management,” *The Journal of Real-Time Systems*, 4, 1992, pages 185-202.

[12] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer, “An Object-Oriented Real-Time Programming Language,” *IEEE Computer*, October 1992, pages 66-73.

[13] Itasca Systems, Inc., *ITASCA System Overview*, Unisys, Minneapolis, Minnesota, 1990.

[14] Kevin B. Kenny, and Kwei-Jay Lin, “Building Flexible Real-Time Systems Using the Flex Language,” *IEEE Computer*, May 1991, pages 70-78.

[15] Won Kim, et al., “Architecture of the ORION Next-Generation Database System,” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pages 109-124.

[16] Victor B. Lortz, “An Object-Oriented Real-Time Database System for Multiprocessors,” *Ph.D. dissertation*, Department of Electrical Engineering and Computer Science, The University of Michigan, March 1994.

[17] Victor B. Lortz, and Kang G. Shin, “MDARTS: A Multiprocessor Database Architecture for Real-Time Systems,” *Technical Report CSE-TR-155-93*, Department of Electrical Engineering and Computer Science, The University of Michigan, March 1993.

[18] S. Marche, “Measuring the Stability of Data Models,” *European Journal of Information Systems*, Vol. 2, No. 1, 1993, pages 37-47.

[19] Martin Marietta Astronautics Group, *Next Generation Workstation/Machine Controller Specification for an Open System Architecture Standard*, NGC-0001-13-000-SYS edition, March 1992.

[20] Clifford W. Mercer, and Hideyuki Tokuda, “The ARTS Real-Time Object Model,” *Proceedings of the 11th Real-Time Systems Symposium*, 1990, pages 2-10.

[21] Simon Monk, and Ian Sommerville, “Schema Evolution in OODBs Using Class Versioning,” *SIGMOD Record*, Vol. 22, No. 3, September 1993, pages 16-22.

[22] Magdi M.A. Morsi, Shamkant B. Navathe, and Hyoung-Joo Kim, “A Schema Management and

Prototyping Interface for an Object-Oriented Database Environment," in F. Van Assche, B. Moulin, and C. Rolland (Editors), *Object Oriented Approach in Information Systems*, Elsevier Science Publishers B.V., 1991, pages 157-180.

[23] Krithi Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases*, 1, 1993, pages 199-226.

[24] J. Penney, and J. Stein, "Class Modification in the GemStore Object-Oriented Database System," *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), October 1987.

[25] Karsten Schwan, Prabha Gopinath, and Win Bo, "CHAOS-Kernel Support for Objects in the Real-Time Domain," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pages 904-916.

[26] Kang G. Shin, and Parameswaran Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *IEEE Proceedings*, Vol. 82, No. 1, January 1994, pages 6-24.

[27] Mukesh Singhal, "Issues and Approaches to Design of Real-Time Database Systems," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pages 19-33.

[28] Dag Sjøberg, "Quantifying Schema Evolution," *Information and Software Technology*, Vol. 35, No. 1, January 1993, pages 35-54.

[29] Hideyuki Tokuda, and Clifford W. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, 23(3), July 1989, pages 29-53.

[30] Ozgur Ulusoy, "Current Research on Real-Time Databases," *SIGMOD Record*, Vol. 21, No. 4, December 1992, pages 16-21.

[31] Victor F. Wolfe, et al., "A Model For Real-Time Object-Oriented Databases," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993, pages 57-63.

[32] Roberto Zicari, "Primitives for schema updates in an Object-Oriented Database System: A proposal," *Computer Standards & Interfaces*, 13, 1991, pages 271-284.