Schema Evolution of an Object-Oriented Real-Time Database System for Manufacturing Automation

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

Abstract— **The database schemata often experience considerable changes during the development and initial use phases of database systems for advanced applications, such as manufacturing automation and computer-aided design. An automated schema evolution system can significantly reduce the amount of effort and potential errors related to schema changes. Although schema evolution for non-real-time databases was the subject of previous research, its impact on real-time database systems remains unexplored. These advanced applications typically utilize object-oriented data models to handle complex data types. However, there exists no agreed-upon real-time object-oriented data model that can be used as foundation to define a schema evolution framework. Therefore, we first design a conceptual real-time object-oriented data model, called ROMPP (*Real-time Object Model with Performance Polymorphism*). It captures the key characteristics of real-time applications, namely, timing constraints and performance polymorphism, by utilizing specialization dimension and letter class hierarchy constructs, respectively. We then re-evaluate previous (non-real-time) schema evolution support in the context of real-time databases. This results in 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—necessary for handling the real-time characteristics of ROMPP. We adopt and extend an axiomatic model to express the semantics of ROMPP schema changes. Using manufacturing control applications, we demonstrate the applicability of ROMPP and potential benefits of the proposed schema evolution system.**

Keywords— **Data model, database, envelope/letter classes, letter class hierarchy, object-oriented, performance polymorphism, real-time, schema evolution.**

I. 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 [6, 8]. Many advanced real-time manufacturing applications, such as open-architecture machine tool controllers, need a database management system (DBMS) to support concurrent data access and provide well-defined interfaces between different software components. These applications typically are subject to a range of timing constraints and often require the DBMS to provide timing guarantees, sometimes under complex conditions. The needs of real-time manufacturing applications in general and machine tool controllers in particular have motivated the work reported in this paper. This research is part of the on-going Open-Architecture Controllers project at the University of Michigan.

L. Zhou and K.G. Shin are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122, USA. E-mail: {lzhou,kgshin}@eecs.umich.edu.

E.A. Rundensteiner is with the Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609-2280, USA. E-mail: rundenst@cs.wpi.edu. The work was performed while the author was at the University of Michigan.

Timing constraints are typically in the form of deadlines. The deadlines of real-time tasks can be classified as *hard*, *firm*, or *soft* [39]. A deadline is said to be hard if the consequences of not meeting it can be catastrophic, such as the deadline of the emergency shutdown task in a machine tool controller. A deadline is firm if the results produced by the corresponding task cease to be useful as soon as the deadline expires, but the consequences of not meeting the deadline are not catastrophic, e.g., the deadline of weather forecast (except for severe weather conditions). 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. An example of soft deadlines may be the deadline of a transaction of an automatic teller machine. The longer the customer waits, the unhappier he or she becomes. Conventional DBMSs do not have any mechanism to specify, and much less to enforce, such complex timing constraints. Furthermore, they do not offer the performance levels or response-time guarantees needed by these real-time applications. Such inadequacy has spawned the field of real-time database systems (RTDBSs) [13, 29, 37, 39, 40, 42, 46].

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 [41], which documents the measurement of schema evolution during the development and initial use of a health management system used at 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 a separate study [22], significant changes (about 59% of attributes on the average) were reported for seven applications. These applications ranged 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, such as engineering design applications using object-oriented databases (OODBs), are typically much less understood and thus are even more prone to changes in the database schemata. In this paper, we investigate the impact of schema evolution for RTDBSs in the context of manufacturing applications.

Machine tool controllers have become more sophisticated in recent years by capitalizing on the progress of computer technology. However, there is still the problem of high lifecycle cost due to the lack of open-ness in commercially-available controllers. There has been considerable interest in the subject, in both academia and industry, North America and Europe. Examples of this activity include the Open System Architecture for Controls within Automation Systems (OSACA) project [33, 34] in the European Community and the Enhanced Machine Controller (EMC) project [1] at the National Institute of Standards & Technology. Research in this area is typically based on the object-oriented paradigm, since it is found to be very suitable for managing real-time data in machine tool controllers.

However, at present no agreed-upon real-time object-oriented data model is available. Thus, we need to define a real-time data model, based on which we can develop a real-time schema evolution framework. Since machine tool controllers are our target applications, our prime objective has been to capture their characteristics. There is a general consensus in the manufacturing community that controllers should have a modular architecture and well-defined interfaces that allow third parties to develop and use these modules independently. Modules can be either hardware or software. A VMEbus-based digital I/O board is an example of a hardware module, while the device driver for the board is an example of a software module. The modules may be selected based on price and/or performance, while meeting the constraints of the control application. An OODB that automates this selection process based on application requirements would be of major help to application developers. This is the main goal of our real-time data model design, namely, to provide facilities for simplifying reuse of time-constrained modules, and consequently, increasing the productivity of real-time application developers and optimizing the utility of resources. None of the existing models used for real-time applications [4, 10, 15, 17, 20, 24, 47] is found to provide sufficient support in this regard.

Based on this observation, we first extract a simple yet powerful real-time object-oriented data model[1] called ROMPP, *Real-time Object Model with Performance Polymorphism* [49]. ROMPP explicitly captures the important characteristics of RTDBS applications, especially in the manufacturing application domain. These characteristics include timing constraints and performance polymorphism.[2] Our model uses two novel constructs–specialization dimensions to model timing specifications and letter class hierarchies to capture performance polymorphism. Although regular object-oriented programming techniques (e.g., composite object classes) may be used to implement the concepts of timing specification and performance polymorphism, they neither explicitly capture these concepts nor provide a mechanism to enforce them. By contrast, ROMPP offers not only explicit constructs for timing specifications but also an automated mechanism to support performance polymorphism.

We then develop a framework for changes to schemata of real-time OODBs based on the schema change taxonomy currently being employed by virtually all existing (non-real-time) schema evolution systems [3]. While schema evolution has been defined for many object-oriented data models [3, 26, 31, 52], none of them is for RTDBSs. We re-evaluate this work in the context of RTDBSs, making 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—necessary for handling additional features specific to the real-time aspects of ROMPP. We use an axiomatic model [32] to formally express the semantics of schema changes. This allows well-defined semantics (as opposed to other schema evolution models that are vaguely described in English language) and easy comparison with other yet-to-be-developed real-time schema evolution approaches. In this paper, we also demonstrate the utility of our real-time object-oriented data model and schema evolution framework based on manufacturing applications.

A preliminary description of ROMPP can be found in [49]. We build upon this research by proposing a schema evolution framework for real-time object-oriented databases in general and for ROMPP in particular. We also present an in-depth evaluation of our approach for machine tool control applications. The main contributions of this paper are summarized below. To our knowledge, schema evolution of RTDBSs has not previously been addressed in the literature.

- Develop a conceptual real-time object-oriented data model, ROMPP.
  - provide constructs for two key characteristics of manufacturing applications–timing constraints and performance polymorphism.
  - allow for explicit annotation of performance metrics of database services.
  - support an automated and transparent mechanism of service selection.
- Propose a schema evolution framework for ROMPP.
  - define new schema change operators.
  - add new schema invariants and resolution rules.
  - uncover and present new semantics of schema changes given real-time constraints, using an extended axiomatic model.
- Demonstrate the applicability of ROMPP and potential benefits of the proposed schema evolution framework in manufacturing applications.

The remainder of the paper is organized as follows. Section II describes ROMPP, while Section III defines a schema evolution framework based on the model. In Section IV, we discuss the implementation status and demon-

[1] Some authors use the terms "object model" and "object-oriented data model" differently. They refer to the object model as the programming model of object-oriented paradigm, and the object-oriented data model as the extension of the programming model in the realm of database management. We make no such distinction and will use the terms interchangeably.

[2] The term *performance polymorphism* first appeared in [17].

strate the utility of our model in the manufacturing control domain. Section V briefly covers related work. Conclusions and future work are presented in Section VI.

II. ROMPP: A CONCEPTUAL REAL-TIME OBJECT MODEL

In this section, we describe our conceptual real-time object model ROMPP. ROMPP 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 by explicitly capturing the key characteristics of real-time applications. In other words, we are not proposing a *complete*[3] data model, but one that is suitable and sufficient for manufacturing applications.

A. Basic Object-Oriented Concepts

ROMPP adopts basic object-oriented concepts, such as class and inheritance, as can be found in most object-oriented data models [7, 11, 18, 27]. For completeness, these concepts are defined below.

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 either a system-provided object or a user-defined object. Instance variables are* **private** *to the object, i.e., they can only be accessed by the object's methods. 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. 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: $M(In_1, In_2, \ldots, In_n) \rightarrow Out$. A parameter specification (either input or output) is a type. 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 itself and other objects.*

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 declarations of common instance variables and methods. The objects of the same class type are called* **instances** *of the class.*

Definition 3. *For two classes C_1 and C_2, C_1 is a* **subclass**[4] *of C_2, denoted by C_1* **is-a** *C_2, if and only if C_1 inherits every instance variable and method of C_2.*

[3] A complete model would have included many more constructs for, for instance, relative temporal consistency among a set of data values and transaction correctness criteria.

[4] Throughout this paper, we say that A is a *subclass* of B (B is a *superclass* of A) if and only if A inherits directly from B, and A is a *descendant* of B (B is an *ancestor* of A) if and only if A inherits directly or indirectly from B.

An example system-provided object is an integer, while a sensor may be a user-defined object. A class can have multiple superclasses. 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 its superclasses are accessible to the subclass and become part of its public interface. 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 (and a private method cannot), i.e., it has no code associated with it and must be implemented in the subclasses (or descendants) of the class.

Definition 4. *A* **class hierarchy** *is a directed acyclic graph (DAG)*[5] *$S = (V, E)$, 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 $e = < C_1, C_2 >$, represents the is-a relationship (C_1 is-a C_2). An* **OODB schema** *is equal to the class hierarchy.*

B. Model Description

Based on our evaluation of existing real-time systems [4, 10, 15, 17, 20, 24, 47] and real-time manufacturing applications [2, 5, 23], we have identified two key characteristics for real-time data models: *timing constraints* and *performance polymorphism*. In open-architecture machine tool controllers, control tasks periodically read sensor data, compute control parameters, and issue actuator commands. All these operations typically need to be completed within each control cycle, i.e., with timing constraints. Open-architecture requirements of machine tool controllers mandate and facilitate the development of hardware and software modules that have the same functionality and interface but with different performance. This characteristic, called performance polymorphism [17], is also a fundamental requirement for manufacturing automation applications. We will show that a simple model capturing these two key characteristics can provide significant help to manufacturing control application developers.

B.1 Timing Constraints

A real-time system must allow the users to specify timing constraints and for the system to enforce them. Any real-time object model must thus have constructs to specify timing constraints. The implementation of a real-time DBMS on the other hand must provide mechanisms to guarantee these deadlines if it is a hard real-time DBMS or make a best effort to meet the deadlines if it is a soft real-time DBMS. The timing constraints of real-time tasks are typically in the form of deadlines.

[5] A class hierarchy without multiple inheritance corresponds to a tree rather than a DAG.

Definition 5. *A* **timing constraint** *is a tuple (***type***,* **description***), where the type specifies the type of the constraint and the description gives the content of the constraint.*

An example of a timing constraint may be (deadline, 10 milliseconds), which specifies a deadline of 10 ms. 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. Applications will be requesting services from objects via their respective methods. We thus need to extend the definition of a method (Definition 1).

Definition 6. *A* **method** *in ROMPP is extended to a triple (***signature***,* **body***[,* **performance***]), with signature and body defined as in Definition 1. The optional third field specifies the performance measures of the method.*

In the above definition, the performance measures include method execution time, memory space needed, and so on. The exact specification of the performance field of a method triple depends on the type of its class, as described in the next subsection.

B.2 Performance Polymorphism

To implement a method, typically several different algorithms and/or data structures can be used. Machine tool controllers need support in selecting one of these implementations based on performance and/or price, by optimizing the objectives of the control applications. For example, a controller for a milling machine may choose among adaptive, linear and nonlinear control algorithms. Although these control algorithms have the same input and output interfaces, they may provide different performance in terms of the quality of generated control commands and the amount of time needed to compute them. The controller may want to select among these different control algorithms based on performance characteristics, but without having to deal with details of their respective implementations. This characteristic of a real-time object model is called performance polymorphism. In type theory, polymorphism is a concept in which a name may denote objects of many different classes that are related via some common superclass ([6], page 102). Performance polymorphism differs from conventional polymorphism in that the distinct characteristics of these related classes are their performance.

Definition 7. **Performance polymorphism** *refers to multiple implementations of a method (body) that carry out the same task but differ in their performance measures.*

Performance polymorphism is explicitly supported by ROMPP, thus allowing an automatic selection of the most appropriate method implementation based on performance characteristics desired by the application. If a real-time object model did not have explicit constructs for performance polymorphism, we would have to use one of the following approaches:

1. The knowledge of performance polymorphism would be captured and maintained separately from the schema. For example, the service designer[6] may use a version control tool to maintain different implementations of the same service (thus having the same service name). The knowledge about such a version control mechanism is not part of the system schema. Although the schema may include a description of different implementations of the service, it provides no help to the application developer on how to use them. Therefore, it is the application developer'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. Furthermore, it does not offer an automated mechanism to ensure the proper use of different implementations of the service. Such approaches do not provide good 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-simplified approach would typically require us to assume a worst-case scenario. This approach may not always be feasible, because requirements may contradict one another. It also wastes resources and poses true limitation on applications. For example, a system may have a memory space of 10MB. Suppose an 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 another implementation of A may need only 2MB to deliver slower but sufficient performance for this particular application.
3. Another option would be 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 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 a mechanism for the application developer to automatically select the most appropriate implementation of a desired service based on requested performance requirements, without having to explicitly choose one of the

[6]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.

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* constructs, which are based on an object-oriented programming technique—the *envelope/letter structure* [9].

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 contains implementation details.*

Definition 9. *A* **letter class hierarchy** *is a class hierarchy as defined in Definition 4 that consists of an envelope class as its root and zero or more letter classes. Each letter class can have exactly one envelope class as its ancestor and no envelope class as its descendant. The envelope class and all its letter classes must have exactly the same public methods.*

In other words, letter classes are all descendants of their corresponding envelope class. While they can have is-a relationships between themselves, these letter classes cannot have is-a relationships with any other envelope class or letter class in different letter class hierarchies. 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. Only envelope classes are visible to the application developer.

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

Definition 11. *A* **real-time object-oriented database (RTOODB) schema** *is composed of one envelope class hierarchy and zero or more letter class hierarchies. Each envelope class can have an optional associated letter class hierarchy and each letter class hierarchy is associated with exactly 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 or 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. A public method of an envelope class can be designated as a *specialization dimension*, as defined below.

Definition 12. *A* **specialization dimension** *is a performance measure (Definition 6) of letter classes. A specialization dimension must be assigned to a public method of an envelope class. 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 may be the execution time of a method. Other examples of specialization dimensions may be the amount of memory needed and the duration the object is valid. The public methods corresponding to a specialization dimension must be declared virtual in the envelope class. This allows the virtual method to be implemented in different ways in the letter classes. A public method can 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. The relative performance of a letter class could be significant in terms of its location in the specialization space. Hence any change to the performance value may map the letter class to a different point 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.

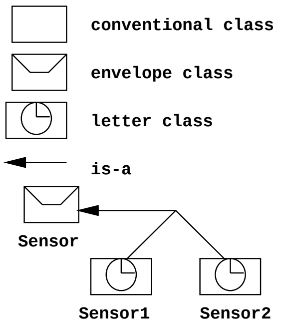
B.3 Model Constructs

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:

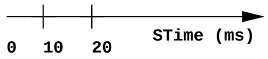
1. `@EC <ec>`
It declares that `<ec>` is an envelope class, where `<ec>` is a class name. This statement is used when defining classes.
2. `@LC <lc> OF <ec>`
It declares that `<lc>`, a class name, is a letter class of the envelope class `<ec>`, again used for class definition.
3. `@DIM: <method> = <identifier>`
It specifies that `<method>`, the name of a method, is a specialization dimension of the letter class hierarchy and gives it a unique identifier (`<identifier>`). This construct can only be used within the definition of an envelope class.

```
// @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) ROMPP schema definition.



(b) Letter class hierarchy.



(c) Specialization space.

Fig. 1. Example of one-dimensional specialization space.

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

It specifies the performance value of the specialization dimension <identifier> that has been declared for its corresponding envelope class. The performance value can be a constant <value>, an expression <expression> (which may use some system-dependent parameters and evaluate to a constant), or a special word unknown. This construct can only be used for letter classes.

Several examples are given below to illustrate 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.

B.4 Examples

Example 1: A Letter Class Hierarchy with One Specialization Dimension

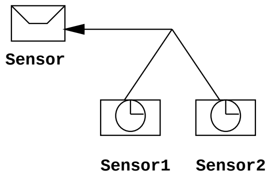
In Figure 1, the class **Sensor** is an envelope class, while classes **Sensor1** and **Sensor2** are its letter classes. The letter classes encapsulate two different implementations of the method **sample()** defined for the envelope class. The method **sample()** has one associated specialization dimension, identified by **STime**. **STime** refers to the requirements on the execution time of the method, and the two letter classes associate different values of the execution time 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

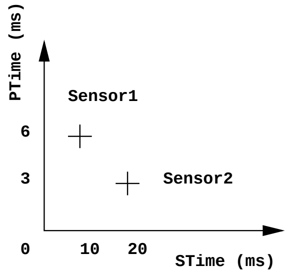
In the example depicted in Figure 2, there are two specialization dimensions, associated with the methods **sample()** and **process()**, respectively. Therefore, the specialization space is a plane, as shown in Figure 2(c). Note that specialization dimensions may not necessarily be inferred

```
// @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) ROMPP schema definition.



(b) Letter class hierarchy.



(c) Specialization space.

Fig. 2. Example of two-dimensional specialization space.

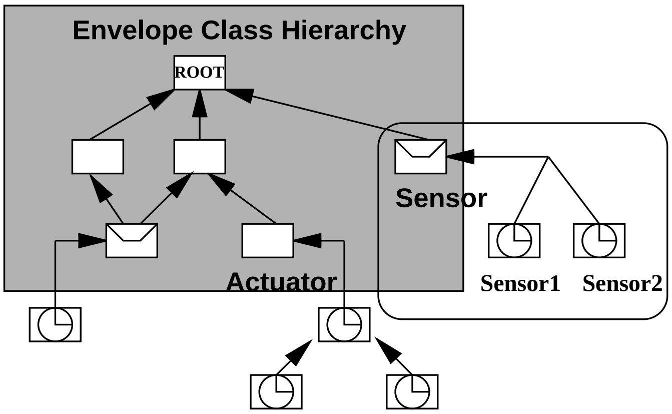


Fig. 3. Example real-time object-oriented database schema.

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.

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.

Assume that an application requires a **Sensor** object with the constraints shown in Figure 4. 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 imple-

```
class Axis {
public:
   ...
private:
   Sensor s("STime<=15ms, PTime<7ms");
   ...
};
```

Fig. 4. Example usage of performance polymorphism.

mentation 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.

III. 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 existing applications (e.g., the class **Axis** in Example 3) should not need any change because of this schema modification. More importantly, our system may direct existing applications to use the newly-added implementation, if it is more appropriate for the specified performance requirements, due to ROMPP's support of performance polymorphism.

Having designed ROMPP, we can now proceed with our task of defining a schema evolution framework for the real-time object model. As we know, there are often several different but all legitimate ways to make a schema change. For example, when deleting a superclass, sometimes it makes sense to keep the inherited attributes of the superclass in its subclasses, while sometimes it does not. There are typically two approaches in dealing with such ambiguity. One is to define rules to completely eliminate any ambiguity. The problem with this approach is that there can be more than one legitimate rule. For example, if a class has multiple superclasses and they have distinct definitions for a single method, it is reasonable for the subclass to use any one of the definitions. Another approach uses no rules and always lets the user decide what to do. The problem with this approach is that it may overburden the user.

Instead, we assume a more balanced approach that integrates the two alternatives. We propose a schema evolution framework that has default rules to resolve any ambiguity of all schema changes, but also allows the user to intervene by confirming or overwriting default rules (before a user-specified action is admitted, the system will check to see if all schema invariants are preserved). For this purpose, we need to apply the typical steps of defining schema evolution framework [3] to our real-time object model as follows:

1. Identify a schema change taxonomy. We need to determine which schema changes are meaningful, given the definition of a ROMPP schema.
2. Identify schema change invariants. In order to keep the consistency of the schema across different modifications, these invariant properties of a ROMPP schema must be preserved.
3. Design schema change rules. When there are alternative ways to do a schema change without violating any invariants, rules are designed to eliminate ambiguity in the context of ROMPP.
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.

A. Schema Change Taxonomy

One of the first object-oriented schema change approaches was proposed by Banerjee *et al.* [3] for ORION [18]. This taxonomy was adopted in most other schema evolution research for OODBs [26, 31, 52] and represents 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 [11, 16, 31] However, this taxonomy does not consider any real-time aspects of object models. We extend the taxonomy with schema changes for ROMPP. A complete list 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) Add a specialization dimension
                  to a method
          (1.2.7) Drop the specialization dimension
                  from a method
(2) Changes to an is-a edge
    (2.1) Make a class a superclass of a class
    (2.2) Remove a class from the superclass
          list of a class
    (2.3) Change the order of superclasses of
          a class
(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
```

Note that there are two additional schema changes, "(1.2.6) Add a specialization dimension to a method" and "(1.2.7) Drop the specialization dimension from a method", which are unique to ROMPP. Although a number of schema changes in our taxonomy are the same as those in [3], we show in Section III-D that the semantics of some of these changes are quite different. In order to support changes of ROMPP schemata, we must evaluate the application of the above types of changes to both letter and envelope class hierarchies.

B. Schema Change Invariants

In order for any schema change 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 following invariants proposed in [3] with some modifications:

1. *Class Hierarchy Invariant.* The class hierarchy is a rooted and directed acyclic graph with uniquely named nodes (classes) and unlabeled edges (is-a 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.
3. *Distinct Origin Invariant.* All same-named methods of a class have distinct origins.[7]
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 ROMPP 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 in the schema and each of them must satisfy the Class Hierarchy Invariant.
7. *Envelope/Letter Class Relationship Invariant.* The declaration of any public method in a letter class must match that in its corresponding envelope class, and *vice versa.*
8. *Specialization Dimension Invariant.* Each specialization dimension has a unique identifier, which is specified for a public method in an envelope class.[8] The identifier is used to reference the corresponding method in the letter classes associated with the envelope class.

[7] Since instance variables are private and invisible to subclasses, they always have distinct origins. So do private methods. Therefore, this invariant refrains an inherited public method from having the same name as a locally-defined method or a public method inherited from a different class.

[8] Not all methods need to be associated with specialization dimensions.

C. Schema Change Rules

We adopt the following schema change rules for ROMPP. They apply to both envelope 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[9] superclass among conflicting superclasses.
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 an existing 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 the 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, either an envelope or a letter class, must be specified for a newly-added letter class.
9. For the deletion of edges from class 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 no longer owns the 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 be also deleted.

The above rules are applicable to ROMPP as well as many non-real-time OODBs. In addition, we identify the following ROMPP-specific rules:

12. Letter classes are dependent on their corresponding envelope classes. That is, deletion of an envelope class will

[9] We assume that superclasses are ordered.

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, such as adding or deleting methods, specialization dimensions, etc., must be propagated to its letter classes. This is to maintain the consistency of the letter class hierarchy 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 the public methods of letter classes is allowed.

D. Schema Change Semantics

All changes to a ROMPP schema can be made to the envelope class hierarchy or letter class hierarchies. 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 both categories of schema changes in this section. Because of the envelope/letter structure of ROMPP, all schema changes to letter class hierarchies and some changes to the envelope class hierarchy have different semantics from traditional ones (e.g., [3]).

It is often dependent on individual applications whether it is meaningful to convert existing instances of a class to those of the modified class. In real-time systems, for example, some objects have only a very short lifetime; thus, it may not be necessary to keep them around after a certain period of time, i.e., no instance conversion. Therefore, we only describe the impact of schema changes on existing data without discussing when and how they are actually converted.

D.1 Axiomatization of Schema Changes

To introduce a formal specification of schema change semantics, we adopt an axiomatic model similar to the one in [32], which has been proven sound and complete. The main differences between ours and that in [32] are:

- We use the terminology of class, subclass and superclass (descendant and ancestor), instead of type, subtype, and supertype.
- The Axiom of Pointedness is not used, since there is no single class used as a common *base* class in ROMPP (i.e., there is not a single class that is the descendant of all classes).
- Private methods and instance variables of superclasses are not visible to subclasses.
- Immediate superclasses of a class t, $P(t)$, are ordered.

The notation of the axiomatic model is shown in Table I. In ROMPP, a class defines properties of objects. The two main properties are instance variables and methods. An instance variable can be further modified by additional properties, such as "name", "private", and "composite". Similarly, methods can have additional properties as well, e.g., "name", "body", "private/public", "specialization dimension", and "performance".

TABLE I

NOTATION OF THE AXIOMATIC MODEL.

Term	Description
T	The lattice of all classes in the schema.
s,t,ROOT	Class elements of T (with s, t variable names and ROOT a special purpose one).
$P(t)$	Ordered list of immediate superclasses of class t.
$P_e(t)$	Ordered list of essential superclasses or ancestors of class t.
$PL(t)$	Superclass lattice of class t.
$N(t)$	Native (local) properties of class t.
$H(t)$	Inherited properties of class t.
$N_e(t)$	Essential properties of class t.
$I(t)$	Interface of class t.
$\alpha_x(f(x), T')$	Apply-all operation ($T' \subseteq T$ and $\forall(x \in T')$, apply the unary function $f(x)$).

T represents all the classes in the schema. $P_e(t)$ are the classes specified by the database designer as essential to the construction of the class t. In other words, $P_e(t)$ should be maintained as superclasses/ancestors of t for as long as possible during schema evolution. The only way to break a link from t to an essential superclass or ancestor s is to explicitly remove s from $P_e(t)$ by either dropping the is-a relationship between s and t or by dropping s entirely. Immediate superclasses $P(t)$ are defined as essential, i.e., $P(t) \subseteq P_e(t)$. The superclass lattice, $PL(t)$, of a class t is the set of all classes of which t is a subclass, including t itself. The native properties, $N(t)$, of a class t are those that are not inherited from any of the superclasses $PL(t)$. The inherited properties, $H(t)$, of a class t are the union of all properties defined by all superclasses of t. The native and inherited properties are disjoint (because of the distinct name and origin invariants), i.e., $N(t) \cap H(t) = \phi$. The essential properties, $N_e(t)$, are those specified by the database designer as essential to the construction of the class t. They consist of all native and possibly some inherited properties, i.e., $N(t) \subseteq N_e(t)$. The interface, $I(t)$, of a class t is the union of native and inherited properties of t, i.e., $I(t) = N(t) \cup H(t)$. The apply-all operation, $\alpha_x(f, T')$, applies the unary function f, over the single variable x, to the elements of a set of classes $T' \subseteq T$. The semantics of the apply-all operation is to let x range over the elements of T' and for each binding of x, evaluate f and include the result in the final result set. For example, the inherited properties $H(t)$ of a class t are the union of the interfaces of its immediate superclasses $P(t)$, i.e., $H(t) = \bigcup \alpha_x(I(x), P(t))$.

Table II shows how various arrangements of classes and properties in Table I can be computed from $P_e(t)$ and $N_e(t)$, which are specified by the database designer. The axioms provide a consistent and automatic mechanism for re-computing the entire class lattice after a change is made to either the essential superclasses $P_e(t)$ or the essential properties $N_e(t)$ of a class t. These schema change axioms are sound and complete [32].

TABLE II

Axioms of subclassing and property inheritance.

Axiom	Description
Closure	$\forall t \in T, P_e(t) \subseteq T$
Acyclicity	$\forall t \in T, t \notin \bigcup \alpha_x(PL(x), P(t))$
Rootedness	$\exists ROOT \in T, \forall t \in T \mid ((ROOT \in PL(t)) \wedge (P_e(ROOT) = \phi))$
Superclasses	$\forall t \in T, P(t) = P_e(t) - \bigcup \alpha_x(PL(x) \cap P_e(t) - \{x\}, P_e(t))$
Superclass Lattice	$\forall t \in T, PL(t) = \bigcup \alpha_x(PL(x), P(t)) \cup \{t\}$
Interface	$\forall t \in T, I(t) = N(t) \cup H(t)$
Nativeness	$\forall t \in T, N(t) = N_e(t) - H(t)$
Inheritance	$\forall t \in T, H(t) = \bigcup \alpha_x(I(x), P(t))$

In what follows, we describe the semantics of schema changes for both letter and envelope class hierarchies in details.

D.2 Schema Changes to a Letter Class Hierarchy

(1) Changes to the contents of a node (a class)

These changes do not modify the topology of the schema. Therefore, in general, only the interface (I) and inherited properties (H) of the affected subclasses need to be recomputed.

(1.1) Changes to an instance variable

Changes to an instance variable can be further divided as follows.

(1.1.1) Add a new instance variable v to class t
$N_e(t) = N_e(t) + \{v\}$;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

The instance variable v is added to the essential properties of the class t. And the descendants of the class are informed of the change in order to adjust their memory allocation.[10] 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 by itself seldom 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.[11] This change affects existing instances of the class.

(1.1.2) Drop an existing instance variable v from class t
$N_e(t) = N_e(t) - \{v\}$;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

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. A software tool to help identify the dependencies will be very useful. The descendants of the class are informed of the change in order to adjust their memory allocation. This change by itself seldom 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 v of class t
$N_e(t) = N_e(t) - \{name(v)\}_{old} + \{name(v)\}_{new}$, where $name(v)$ denotes the name of the instance variable v, and the subscripts "old" and "new" denote the old and new values, respectively;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

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 v of class t
$N_e(t) = N_e(t) - \{v\}_{old} + \{v\}_{new}$;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

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). It could have the same impact on method performance as in (1.1.1). This change affects existing instances of the class.

(1.1.5) Drop the composite property of an instance variable v from class t
$N_e(t) = N_e(t) - \{composite(v)\}$, where $composite(v)$ denotes the composite property of the instance variable v;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

Rules 10 and 11 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[12]

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. However, some changes are not allowed for letter classes (see below).

(1.2.1) Add a new method m to class t
```
if (m is public) {
    reject;
} else {
    Ne(t) = Ne(t) + {m};
    ∀(s ∈ T) ∧ (t ∈ PL(s)), recompute I(s), H(s);
}
```

If the new method is public, the change is not allowed unless it is initiated by the corresponding envelope class (Rule 14). In this case, the change is made to the root of the letter class hierarchy and then propagated to all letter

[10] Properties of the class s that are not recomputed, e.g., $P(s)$, $P_e(s)$, $PL(s)$, $N(s)$ and $N_e(s)$, are not affected.

[11] If the specialization space is organized as a linked list, as mentioned in Section II-B, no reorganization will be needed. If it is organized as an ordered list, then the node representing the affected letter class will have to be re-inserted in the correct position of the list, depending on the new performance value.

[12] The impact of schema changes on behaviors of objects is referred to as the behavior consistency problem in [52].

classes (Rule 13). Such a change may affect the specialization space, if the new method represents a new specialization dimension. If the change causes any conflict, 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, because the new method has not been used yet.

(1.2.2) Drop an existing method m from class t

if (m is public) {
  reject;
} else {
  $N_e(t) = N_e(t) - \{m\}$;
  $\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;
}

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 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 m of class t

if (m is public) {
  reject;
} else {
  $N_e(t) = N_e(t) - \{name(m)\}_{old} + \{name(m)\}_{new}$;
  $\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;
}

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. This change does not affect the specialization space and existing instances. Of course, all references to the old method name must be updated.

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

$N_e(t) = N_e(t) - \{body(m)\}_{old} + \{body(m)\}_{new}$;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

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.[13] 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 III-D.4. 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 m in class t

$N_e(t) = N_e(t) - \{m\}_{old} + \{m\}_{new}$;
$\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;

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.

[13] The performance can be either analyzed and determined empirically by the service designer or evaluated by an automated analysis system.

(1.2.6) Add a specialization dimension to a method m in class t

if (t is not an envelope class) {
  reject;
} else {
  $N_e(t) = N_e(t) + \{dim(m)\}$, where $dim(m)$ denotes the specialization dimension property of the method m;
  $\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;
}

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 from a method m in class t

if (t is not an envelope class) {
  reject;
} else {
  $N_e(t) = N_e(t) - \{dim(m)\}$;
  $\forall (s \in T) \wedge (t \in PL(s))$, recompute $I(s), H(s)$;
}

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 is-a edge

These changes, in general, modify the topology of the schema. Therefore, not only the interface (I) and inherited properties (H) but also the ordered list of immediate superclasses (P) of the affected subclasses may need to be recomputed.

(2.1) Make a class s a superclass of class c

if (c is not a letter class || s is not in the same letter class hierarchy) {
  reject;
} else if ($s \notin P_e(c)$){
  $P_e(c) = P_e(c)$ appended with s;
  recompute $I(c), H(c), P(c)$;
  $\forall (t \in T) \wedge (c \in PL(t))$, recompute $I(t), H(t), P(t)$;
}

Class c must be a letter class and s must be in the letter class hierarchy associated with c. Class s is made the last one in c's superclass list. Class c now inherits additional public methods from s, if any. Any signature conflicts may be ignored since s is the last of c's superclasses. This schema change may change c's position in the specialization space and the space may thus need to be reorganized. This change affects existing instances of the class c.

(2.2) Remove a class s from the superclass list of class c

if (s is an envelope class) {
  reject;
} else {
  $P_e(c) = P_e(c) - \{s\}$;
  recompute $I(c), H(c), P(c)$;
  $\forall (t \in T) \wedge (c \in PL(t))$, recompute $I(t), H(t), P(t)$;
}

Class 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 (Rule 12). 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 class c
reorder $P_e(c)$;
recompute $I(c), H(c), P(c)$;
$\forall(t \in T) \wedge (c \in PL(t))$, recompute $I(t), H(t), P(t)$;

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 of that in s_1. This change affects existing instances of the class c.

(3) Changes to a node

(3.1) Add a new class c (as the subclass of class s)
create c;
$P_e(c) = P_e(c) + \{s\}$;

An envelope class or a letter class must be specified as its parent (Rule 8). It adds a new point in the specialization space. The new class has no instances.

(3.2) Drop an existing class s

```
if (s is an envelope class) {
    reject;
} else {
    remove s;
    ∀(c ∈ T) ∧ (s ∈ P(c)), do (2.2);
}
```

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 class s
rename s;
$\forall(c \in T) \wedge (s \in P_e(c)), P_e(c) = P_e(c) - \{name(s)\}_{old} + \{name(s)\}_{new}$;

This change does not affect the specialization space. It may require s's subclasses to change their parent's name.

D.3 Schema Changes to an Envelope Class Hierarchy

In general, changes to an envelope class hierarchy have similar semantics to those defined in [3]. In addition, the changes must be propagated to the corresponding letter classes, if any, since letter class hierarchies are descendants of their corresponding envelope classes. The changes 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. 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 s

```
∀(c ∈ T) ∧ (s ∈ PL(c)) do {
    if (c is in the letter class hierarchy associated with s) {
        drop c;
    }
}
```

Drop the class s 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 ROMPP. 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) Add a specialization dimension to a method m of envelope class s

```
if (m is not public) {
    reject;
} else {
    N_e(s) = N_e(s) + {dim(m)};
    ∀(c ∈ T) ∧ (s ∈ PL(c)), recompute I(c), H(c);
}
```

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 Section III.D-4). The specialization space has one more dimension now and may need to be reorganized.

(1.2.7) Drop the specialization dimension from a method m of envelope class s
$N_e(s) = N_e(s) - \{dim(m)\}$;
$\forall(c \in T) \wedge (s \in PL(c))$, recompute $I(c), H(c)$;

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

D.4 Example of Schema Changes for a Manufacturing Database

Suppose we have the following letter class hierarchy (Figure 5), 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 is to add a new specialization dimension, **PTime**, to the method **process()** using the command "`ADD DIM PTime TO void process() IN Sensor`". According to the semantics defined for schema change (1.2.6) in Section III-D.2, 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** (Figure 6), which causes the addition of new performance measures associated with all occurrences of the **process()** method (line 16 and line 26). Since the

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

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

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

Fig. 5. An example of schema changes.

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

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

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

Fig. 6. After adding a new specialization dimension.

system does not know the performance of the method **process()** in letter classes yet, it puts **unknown** there. 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 the **Sensor1** class, 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. The performance analysis may be done by the application developer, either by code analysis or by calibration experiments. Suppose the worst-case execution time for this particular implementation of **process()** in **Sensor1** is 6 ms, then the system modifies the performance measure associated with the method as in line 16 of Figure 7. The schema evolution mechanism ensures that performance information of letter classes is up-to-date. This is essential in order for the requirements-driven automatic implementation selection mechanism to work.

```
11 // @LC: Sensor1 OF Sensor
12 class Sensor1 : public Sensor {
13 public:
14   // @DIM: STime = 10 ms
15   int sample();
16   // @DIM: PTime = 6 ms
17   void process();
18   ...
19 };
```

Fig. 7. After changing the code for **process()** in **Sensor1**.

IV. Implementation Status

A. *System Overview*

We have built a prototype of an object-oriented RTDBS for machine tool controllers. Since existing commercial OODBs do not meet the performance level and predictability required for our target domain, we had to build our RTDBS from scratch. The system architecture is illustrated in Figure 8. The RTDBS incorporates ROMPP as a foundation. Modules used in our RTDBS include reusable libraries, and tools for performance evaluation and schema evolution. Different implementations of database services (letter classes) are organized in reusable class libraries. These libraries also include many useful system classes, such as a class called **Task**. It is a generic real-time task and can be used to compose application tasks [50]. The RTDBS uses performance evaluation tools to measure, analyze, and store performance information of database services and application modules. Schema evolution tools will help the user make changes to the application.

Our initial implementation effort has focused on developing the object manager supporting the real-time object model ROMPP defined in Section II and the underlying database services. This is because we need to gain more first-hand experience of developing control applications using our RTDBS before implementing the schema evolution framework. To demonstrate the utility of ROMPP and indicate the potential of adding the proposed schema evolution framework, we use real-time machine tool controllers to discuss the implementation of our RTDBS. Our system is currently being utilized by researchers in the Department of Mechanical Engineering and Applied Mechanics at the University of Michigan for controlling a 5-axis milling machine.

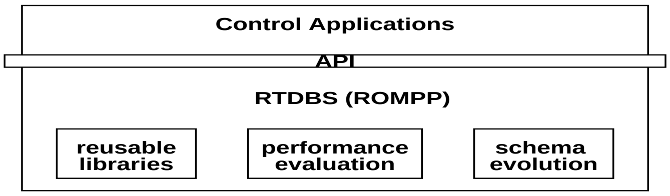


Fig. 8. System architecture.

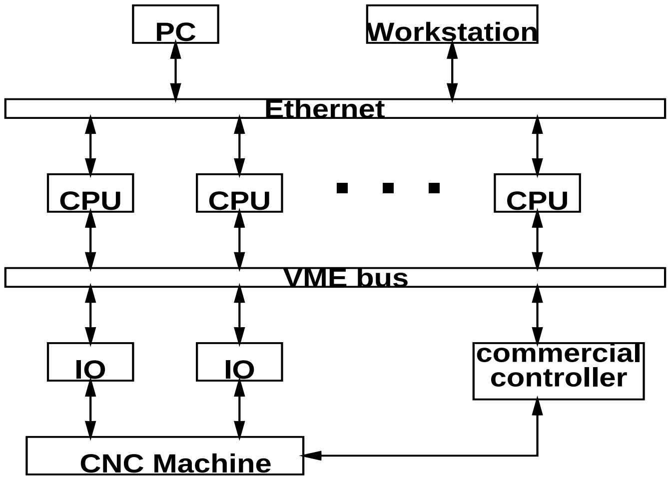


Fig. 9. UMOAC testbed.

B. UMOAC Testbed

Before discussing implementation details of the RTDBS software, we need to describe its hardware setup. Different hardware configurations may result in different software implementation strategies. Our prototype RTDBS and control applications are being developed on the University of Michigan Open-Architecture Controller (UMOAC) testbed (Figure 9). Control tasks, as well as the RTDBS, run on VMEbus-based processor boards–CPUs in the figure (e.g., Motorola 680x0s or Intel x86s). In order to achieve good performance and timing predictability, a real-time operating system (e.g., VxWorks or QNX) is used for these processors. Sensors and actuators on the computer numerically-controlled (CNC) machine are accessed through commercial controllers (e.g., Delta Tau PMAC) and/or IO interface boards (e.g., XYCOM XVME 201 digital IO board). Control software may be cross-developed on, and downloaded from, remote computers connected to the testbed via Ethernet. This testbed architecture allows easy adoption of new hardware components as they become available, and thus provides good hardware openness. Well-defined interfaces and support for performance polymorphism will provide a foundation of software openness.

C. Application-Programming Interface (API)

Our RTDBS offers a unique application-programming interface (API) for manufacturing applications. The API allows the user to explicitly specify timing constraints. The constraints are described in a string,

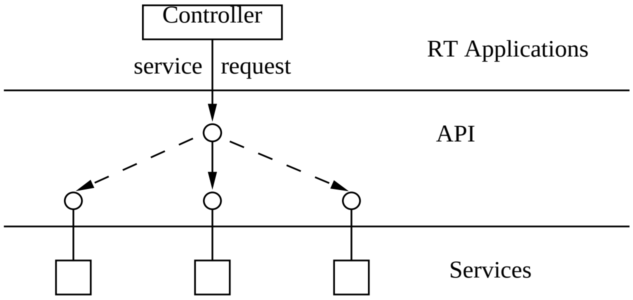


Fig. 10. Application-programming interface for manufacturing.

thus making it easy to add new constraint types in the future. The API is similar to that in Figure 4, e.g., "**Sensor s("position_sensor", "exclusive_update;write<=50usec", CREATE);**". The first argument is the name of the sensor, while the second captures the constraints on the sensor object. The last argument indicates that the object should be created if it does not exist. The API provides the user access to the underlying ROMPP services, including an automated mechanism that selects software modules based on application requirements, as visualized in Figure 10. For example, the service designer provides a collection of system services that constitute the kernel of the RTDBS. When a machine tool controller (built by the application developer) needs some service, it sends the RTDBS a service request, which specifies the type of service, performance constraints, and other requirements, using the API. The RTDBS, which supports ROMPP, will automatically select the most appropriate service for the request. This selection process may be accomplished either at application start-up time or at run-time.

D. ROMPP

ROMPP uses specialization dimensions to characterize timing constraints and letter class hierarchies to capture performance polymorphism. Constraints are specified in a string, which is then passed to the part of RTDBS that implements performance polymorphism. An exemplar-based technique [9] is adopted to realize the automatic performance polymorphism mechanism. This mechanism customizes applications by binding appropriate service objects with the applications at their start-up time, according to their respective constraints. Exemplars are special, one-per-class objects that are prototype representatives of an entire class. Given an exemplar object, applications can construct copies of the exemplar by invoking a special method. These copies (corresponding to the instances of letter classes in ROMPP) represent different implementations of the base class (corresponding to the envelope class in ROMPP). The exemplars (letter classes) are organized as a list, sorted by decreasing performance values of a chosen specialization dimension. The first one that satisfies all requirements will be used by the application.[14]

[14] A simple linear organization of letter classes may not be optimal when there are multiple specialization dimensions. More research is

Rather than choosing a specific implementation class, an application chooses a base class and specifies the rest of its requirements in a string, which is passed to the population of implementation classes (exemplars) derived from that base class. The exemplars then examine the application requirements. An object that meets all requirements is returned to the application. Since all exemplars (letter classes) support the same functional interface, applications can use the returned object without knowledge of which class was actually constructed. Therefore, applications need only contain dependencies on the abstract base classes (envelope classes). This permits letter classes to be extended, modified, and reorganized without requiring corresponding modifications to existing applications.

A number of database service objects, organized as a class library, have been implemented to facilitate the development of control applications. For example, a task template may be used as a building block for a periodic control task [50]. In order to achieve high performance, database transactions are embedded in application tasks and executed directly in main memory, as opposed to in the database server, thus avoiding the context switching cost. Data integrity and concurrency control are supported since all data accesses are made through the transaction methods exported by the database service objects. The object-oriented approach is thus critical to our system. This transaction execution model maximizes the performance benefits of using main memory because it eliminates the overhead implicit in client-server architectures [20].

E. Performance Evaluation

A key problem in utilizing ROMPP for a particular application is how to obtain performance values of methods for a specialization dimension, in particular, method execution times. One might think that it would be easy to determine the execution time of a method by analyzing its source code. Unfortunately, this is a variant of the famous Turing machine halting problem, which is in general undecidable [14]. If restrictions are placed on the code, such as prohibiting loops and carefully controlling I/O, it becomes theoretically possible to synthesize the execution time.

However, with modern CPU architectures that employ caches and pipelines, this analysis of method execution times can be very difficult. Therefore, we are pursuing an experimental approach to determine execution times. For example, to obtain the worst-case response time for a shared object access (read or write) operation with a known maximum number (say n) of concurrent access operations for that particular shared object, we can run these n operations on multiple processors in parallel and measure their execution times. The longest response time of an operation will occur when all n operations are released at the same time and this particular operation gets executed last. Alternatively, these n operations can be run sequentially. The worst-case response time will be the elapsed time between the release of the first operation and the completion of the last operation.

Given a clock with a fine resolution and methods that exhibit predictable performance (which is the case for our real-time applications), we believe that the experimental approach is sufficient to characterize worst-case response times. Tools for performance evaluation are being developed and will be integrated into ROMPP. The preliminary results of our empirical studies are reported in [51].

needed on this issue.

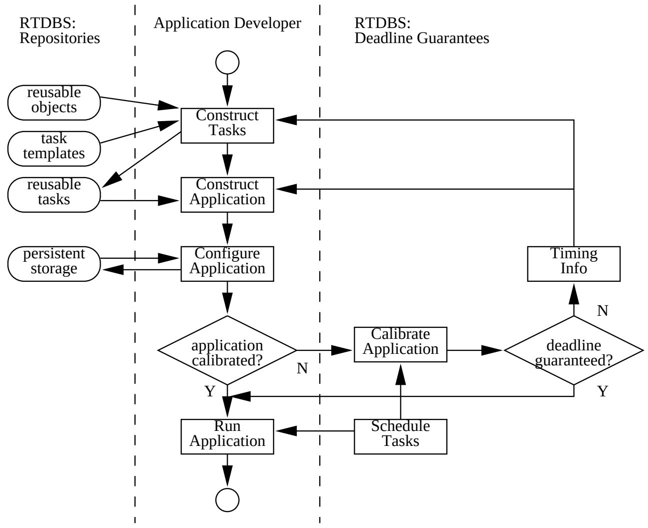


Fig. 11. Application deadline guarantees using RTDBS.

F. Need for Schema Evolution Support

Figure 11 illustrates how our RTDBS can be used in control application development. The application developer first constructs individual control tasks using task templates provided by the RTDBS and reusable objects previously developed and stored in the RTDBS. These control tasks may also be reusable modules for future applications. In the next step, the application is configured with machine specifications (e.g., work table dimensions and velocity limits) and control parameters (e.g., gains). These data may be saved in the persistent storage, so that when the application runs again, it only needs to retrieve the data from the storage. If the deadlines of the application have not been checked and guaranteed by the RTDBS, the application goes into the calibration phase. The RTDBS checks timing constraints during calibration, using the performance measurement tools described earlier. If all deadlines can be guaranteed, the RTDBS "accepts" the application (i.e., guarantees all its deadlines). If not, the RTDBS provides timing information so that the user can modify the tasks or the application configuration. All accepted applications can run without any further calibration.

It is obvious that, in the development of a control application, the application may require numerous changes before its functionality and performance can meet all constraints. This clearly indicates a need for the support of a schema evolution framework. Building such support will be evolutionary, and we have established a solid foundation for the real-time schema evolution framework. We will incor-

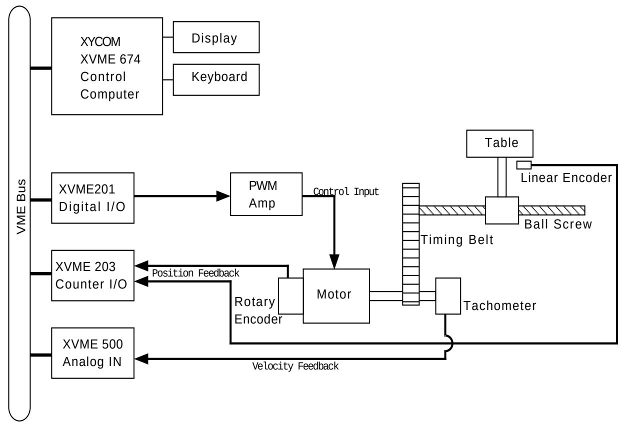


Fig. 12. Hardware setup of a three-axis controller.

porate the schema evolution framework into our RTDBS as we gain more experiences with a variety of manufacturing applications.

G. Prototype Three-Axis Controller

To evaluate the suitability of our RTDBS in the domain of real-time manufacturing control applications, a prototype three-axis milling machine controller was developed. Figure 12 shows the hardware setup (only one axis is shown, since the other two are similar). There are five main tasks in this application (Figure 13). The Graphical User Interface (GUI) task is aperiodic. It allows the user to enter control commands and it displays application information. The Human-Machine Interface (HMI) task runs every 100 ms at a priority of 24.[15] It checks for any command that may be sent by the user via the GUI task and dispatches appropriate commands to the X-, Y- and Z-Axis control tasks, which run every 10 ms at a priority of 26. The HMI, X-Axis, Y-Axis and Z-Axis tasks are run on the XYCOM XVME 674, a VMEbus-based 66MHz 486DX2 with 32MB of RAM running the QNX real-time operating system (see Figure 12). Since these tasks are real-time tasks, the communication among them is via shared memory in order to minimize run-time overhead. The X-Axis task uses the XVME 203 Counter I/O board to get the position of the X axis from the rotary and linear encoders, and uses the XVME 500 Analog Input board to get the velocity of the X axis from the tachometer. The Y- and Z-Axis tasks have a similar setting. The GUI task runs on a different IBM compatible PC connected to the XYCOM 674 via Ethernet (not shown in Figure 12). The GUI task communicates with the HMI task using message passing.

The control functionality is performed by the X-, Y- and Z-Axis control tasks. To control the motion of each axis, either the PID control algorithm or the fuzzy logic control algorithm (developed by mechanical engineers) [28] is used. In the PID control, a position error (difference between a reference position input and a feedback from an encoder) and velocity feedback from a tachometer (at current and previous time steps) are used as inputs. In the fuzzy logic control, a position error and a change in the position errors between the last two time steps are the inputs. A control command is calculated using the respective control algorithm. It is then sent to a PWM board through a digital I/O board (XVME 201 in Figure 12).

[15]These tasks are run under QNX, which supports priority-based preemptive scheduling. A larger number represents a higher priority.

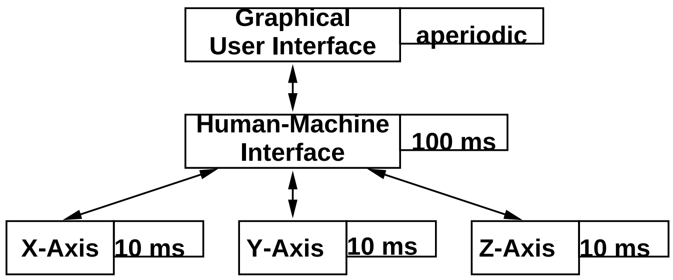


Fig. 13. Tasks of the 3-axis control application.

There are two specialization dimensions in the axis control tasks: **ExecutionTime** and **ControlLaw**. They represent the characteristics for which the implementations of axis control tasks may differ. The **ExecutionTime** dimension corresponds to the elapsed time from start to end of the task execution, which includes any time the task is blocked or preempted. The **ControlLaw** dimension currently has two values: **stability** and **accuracy**. In our experiments, the controlled machining process is found sometimes to become unstable when the fuzzy logic control algorithm is used. By contrast, the controlled process is very stable with the PID control algorithm. Therefore, when the stability is important the PID control algorithm is used, and when accuracy is emphasized the fuzzy logic control algorithm is chosen. This selection is done during the application initialization in order to eliminate run-time overhead (the appropriate implementation of the axis control task is selected automatically at the application startup time based on the requirements). This method of binding objects at initialization time is essential for meeting the needs of manufacturing applications, because it provides as small a response time as tens of microseconds required by the application.

Without the automated mechanism of performance polymorphism explicitly supported by ROMPP, the application developer would have to figure out exactly which database services to use. Whenever the application requirements and/or database service implementations change, the application developer has to find suitable services again and modify the application code accordingly. With the automated mechanism, all the application developer needs to do is to change the requirement specifications (in the case of application requirement changes) or nothing (in the case of service implementation changes). The system will take care of the service selection and binding. The prototype controller was effectively utilized to cut metal parts using the milling machine.

V. RELATED WORK

A. Schema Evolution

There has been considerable work on defining schema evolution for OODBs. Examples are schema evolution for ORION [3], O2 [52], GemStone [31] and GOOSE [26]. However, none of them addressed schema evolution in the context of real-time OODBs. The traditional approach is to define a number of invariants that must be satisfied by the schema and then to define rules for maintaining these invariants (e.g., [3]). To avoid expensive changes to existing applications dependent on the original schema, researchers studied other approaches such as object-oriented views [35, 36, 44] and versions [19, 25]. [32] proposes an axiomatic model to provide a common framework for defining and comparing different schema evolution policies. All of these typically support the schema change taxonomy initially proposed for ORION [3]. In this paper, we focus instead on the real-time aspects of schema evolution, in addition to the traditional schema change taxonomy. To our knowledge, this work in schema evolution of real-time object-oriented databases is the first of its kind.

B. Real-Time Models

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 of real-time applications and should be explicitly supported by a real-time data model.

CHAOS (Concurrent Hierarchical Adaptable Object System) [4, 38] 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) [24, 45] is a distributed real-time operating system kernel. RTC++ [15] 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, 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 to remedy this. As discussed in Section II-B.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 [17] 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) [10] 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, though it supports *contingency plans*. Contingency plans are alternate actions that can be invoked whenever the system determines that it cannot complete a task in time. Examples of contingency plans are the use of less resolution in a spatial search or the use of old aggregate data if the aggregate changes only slowly in response to updates to underlying data. Contingency plans are closer to the concept of imprecise computation, which mainly deals with the deadline constraint by sacrificing the quality of results. HiPAC 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.

RTSORAC (Real-Time Semantic Objects Relationships And Constraints) [12, 30, 47] incorporates a comprehensive model for concurrency control in real-time OODBs and a flexible approach to synchronizing real-time transactions. It considers a broad range of semantic information regarding logical and temporal consistency, and allows a wide range of correctness criteria that relax serializability. However, performance polymorphism is again not provided.

MDARTS (Multiprocessor Database Architecture for Real-Time Systems) [20, 21] 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. 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.

VI. Conclusions and Future Work

In this paper, we proposed solutions to the schema evolution problem for real-time OODBs. Schema evolution support is becoming increasingly important, as advanced real-time applications are starting to demand database services, rather than *ad hoc* data repositories. This demand comes from the needs 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 in revamping an existing system based on changes of technology and/or environment. They also need support to quickly configure new customized systems.

We identified timing constraints and performance polymorphism as two key characteristics of real-time manufacturing applications. We presented a conceptual real-time object model, ROMPP, which explicitly captures these two features. It uses specialization dimensions to model timing constraints and utilizes letter class hierarchy constructs to capture performance polymorphism. In the context of RTDBS, we proposed 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. Using manufacturing control applications, we demonstrated the applicability of ROMPP and potential benefits of the proposed schema evolution system.

There are still several open questions to be answered. In particular, we need to improve the utilization of computational resources when hard deadline guarantees are relaxed to probabilistic deadline guarantees. We have observed that the worst-case execution time can be much longer than the average. Figure 14 shows the histograms of the axis control task execution times. For example, among the 658 samples of the Y-Axis task execution time, all are below 1.20 ms except two samples. One of them is about 8.98 ms and the other is about 8.06 ms. To provide hard deadline guarantees, we have to use the worst-case execution times. However, there are often situations where the deadline can be missed once in a while. For example, a sensor-reading task typically computes the average of several readings. If it occasionally misses the deadline, there will not have much impact on the average. Obviously, if the worst-case execution time is used for the scheduling of such tasks, it can waste a significant amount of computational resources. In this case, it may be appropriate to introduce the notion of *completion probability*, which specifies the required probability that a task must meet its deadline. These tasks require *probabilistic* deadline guarantees. This is one of the issues we are currently investigating [51].

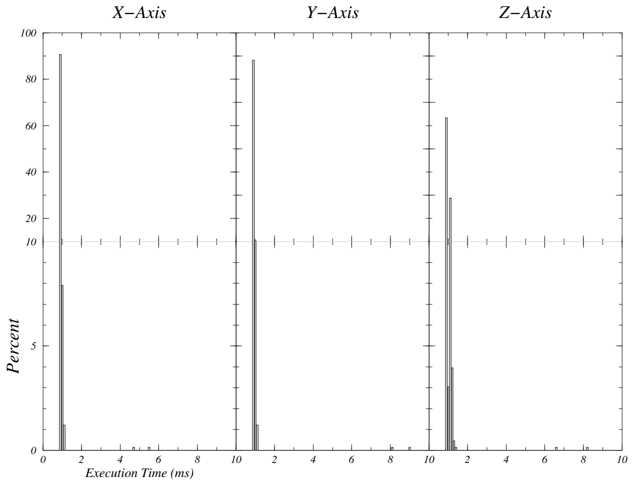


Fig. 14. Histograms of the measured execution times of the axis control tasks.

We would also like 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). The results reported here are a good first step to explore the area of schema evolution for RTDBSs, and will spawn new research efforts.

Acknowledgments

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 a Graduate Fellowship, and the National Science Foundation under grants DDM-9313222 and IRI-9504412. The views and conclusions contained in this document are those of the authors and should not be interpreted as that of the sponsors, either expressed or implied.

We would also like to thank Nauman A. Chaudhry, Matthew C. Jones, Harumi A. Kuno, Amy Lee, and Young-Gook Ra, for critiquing this work.

References

[1] J. Albus, "NIST Open-Architecture," *Int'l Workshop on Open-Architecture Controllers for Automation*, April 1994.

[2] B. Anderson, "Next Generation Workstation/Machine Controller (NGC)," *Proc. IPC'92*, April 1992, pp. xix-xxvi.

[3] J. Banerjee, W. Kim, H.-J. Kim, and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *SIGMOD*, 1987, pp. 311-322.

[4] T. E. Bihari, and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples," *IEEE Computers*, December 1992, pp. 25-32.

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

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

[7] P. Butterworth, A. Otis, and J. Stein, "The Gemstone Object Database Management System," *Communications of the ACM*, Vol. 34, No. 10, October 1991, pp. 64-77.

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

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

[10] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari, "The HiPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 51-70.

[11] O. Deux, et al., "The O2 System," *Communications of the ACM*, Vol. 34, No. 10, October 1991, pp. 34-48.

[12] L. DiPippo, and V. Wolfe, "Object-based Semantic Real-time Concurrency Control," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 87-96.

[13] M.H. Graham, "Issues in Real-Time Data Management," *Journal of Real-Time Systems*, 4, 1992, pp. 185-202.

[14] J.E. Hopcroft, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[15] Y. Ihikawa, H. Tokuda, and C.W. Mercer, "An Object-Oriented Real-Time Programming Language," *IEEE Computer*, October 1992, pp. 66-73.

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

[17] K.B. Kenny, and K.-J. Lin, "Building Flexible Real-Time Systems Using the Flex Language," *IEEE Computer*, May 1991, pp. 70-78.

[18] W. Kim, J.F. Garza, N. Ballou, and D. Woelk, "Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 109-124.

[19] W. Kim, and H.-T. Chou, "Version of Schema For Object-Oriented Databases," *MCC Technical Report*, ACA-ST-278-87, Rev. 1, February 1988.

[20] V.B. Lortz, "An Object-Oriented Real-Time Database System for Multiprocessors," *Ph.D. dissertation*, Department of Electrical Engineering and Computer Science, University of Michigan, March 1994.

[21] V.B. Lortz, and K.G. Shin, "MDARTS: A Multiprocessor Database Architecture for Real-Time Systems," *Technical Report CSE-TR-155-93*, Department of Electrical Engineering and Computer Science, University of Michigan, March 1993.

[22] S. Marche, "Measuring the Stability of Data Models," *European Journal of Information Systems*, 1993, pp. 37-47.

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

[24] C.W. Mercer, and H. Tokuda, "The ARTS Real-Time Object Model," *Proceedings of the 11th Real-Time Systems Symposium*, 1990, pp. 2-10.

[25] S. Monk, and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning," *SIGMOD Record*, Vol. 22, No. 3, September 1993, pp. 16-22.

[26] M.M.A. Morsi, S.B. Navathe, and H.-J. 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, pp. 157-180.

[27] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1992.

[28] Open-Architecture Controls Team, *Developer's Guide for Open-Architecture Control of the Robotool*, Department of Electrical Engineering and Computer Science and Department of Mechanical Engineering and Applied Mechanics, The University of Michigan, November 1995.

[29] G. Ozsoyoglu, and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995, pp. 513-532.

[30] J. Peckham, V.F. Wolfe, JJ Prichard, and L.C. DiPippo, "RT-SORAC: Design of a Real-Time Object-Oriented Database System," *Technical Report 94-231*, University of Rhode Island, 1994.

[31] 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*, October 1987.

[32] R.J. Peters, and M.T. Ozsu, "Axiomatization of Dynamic Schema Evolution in Objectbases," *Proceedings of the 11th International Conference on Data Engineering*, March 1995, pp. 156-164.

[33] G. Pritschow and G. Junghans, presentations at the *International Workshop on Open-Architecture Controllers for Automation*, Ann Arbor, Michigan, April 1994.

[34] G. Pritschow, and C. Daniel, "Open Control System — A Future-Oriented Concept," *Proceedings of the 27th CIRP International Seminar on Manufacturing Systems*, May 1995, pp. 5-17.

[35] Y.G. Ra, and E.A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Evolution," *Proceedings of the 11th International Conference on Data Engineering*, March 1995, pp. 165-172.

[36] Y.G. Ra, and E.A. Rundensteiner, "A Transparent Schema Evolution System Based on Object-Oriented View Technology," to appear in *IEEE Transactions on Knowledge and Data Engineering*, 1997.

[37] K. Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases*, 1, 1993, pp. 199-226.

[38] K. Schwan, P. Gopinath, and W. Bo, "CHAOS-Kernel Support for Objects in the Real-Time Domain," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pp. 904-916.

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

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

[41] D. Sjoberg, "Quantifying Schema Evolution," *Information and Software Technology*, January 1993, pp. 35-54.

[42] N. Soparkar, H.F. Korth, and A. Silberschatz, "Database with Deadline and Contingency Constraints," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995, pp. 552-565.

[43] A. Stevens, *C++ Database Development*, 2nd ed., MIS:Press, 1994.

[44] V.C. Taube, and E.A. Rundensteiner, "Schema Removal Issues for Transparent Schema Evolution," *Sixth International Workshop on Research Issues on Data Engineering, Interoperability of Nontraditional Database Systems*, Feb. 1996.

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

[46] O. Ulusoy, "Current Research on Real-Time Databases," *SIGMOD Record*, Vol. 21, No. 4, December 1992, pp. 16-21.

[47] V.F. Wolfe, L.B. Cingiser, J. Peckham, and J. Prichard, "A Model For Real-Time Object-Oriented Databases," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993, pp. 57-63.

[48] L. Zhou, E.A. Rundensteiner, and K.G. Shin, "Schema Evolution for Real-Time Object-Oriented Databases," *Technical Report*

CSE-TR-199-94, Department of EECS, University of Michigan, March 1994.

[49] L. Zhou, E.A. Rundensteiner, and K.G. Shin, "OODB Support for Real-Time Open-Architecture Controllers," *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications*, April 1995, pp. 206-213.

[50] L. Zhou, M.J. Washburn, K.G. Shin, and E.A. Rundensteiner, "Performance Evaluation of Modular Real-Time Controllers," to appear in the *1996 ASME International Mechanical Engineering Congress and Exposition*, November 1996.

[51] L. Zhou, K.G. Shin, E.A. Rundensteiner, and N. Soparkar, "Probabilistic Real-Time Data Access with Deadline and Interval Constraints," to appear in Sang H. Son, Kwei-Jay Lin and Azer Bestavros ed., *Real-Time Databases Systems: Issues and Applications*, Kluwer Academic Publishers, 1997.

[52] R. Zicari, "Primitives for schema updates in an Object-Oriented Database System: A proposal," *Computer Standards and Interfaces*, 13, 1991, pp. 271-284.

Lei Zhou is a Ph.D. candidate in computer science and engineering at the University of Michigan, Ann Arbor, where he has been working on real-time object-oriented data management for manufacturing applications. His interests include software research and development in databases, real-time systems, VLSI CAD and telecommunications.

He received a B.S. and an M.S. from Fudan University, Shanghai, China, in 1986 and 1989, respectively, both in electrical engineering, and an M.S. in computer science and engineering from Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, in 1991.

Elke Angelika Rundensteiner received a BS degree (Vordiplom) from the Johann Wolfgang Goethe University, Frankfurt, Germany, in 1984, a Master's degree from the Florida State University, Tallahassee, in 1987, and her Ph.D. degree from the University of California, Irvine, in 1992; all in Computer Science. Dr. Rundensteiner has received numerous honors and awards, including a Fulbright Scholarship, an NSF Young Investigator Award in databases in 1994, and an Intel Young Investigator Engineering Award from the Engineering Foundation. Dr. Rundensteiner is member of IEEE and ACM.

She recently joined the Department of Computer Science at the Worcester Polytechnic Institute, after having been an Assistant Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. Dr. Rundensteiner has been investigating database technology for non-conventional applications for over ten years. Her research interests include object-oriented view techniques for data warehousing and database evolution, database tools for digital library and electronic commerce applications, multimedia databases, and geographic information systems.

Kang G. Shin is Professor and Director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan.

He has authored/coauthored over 360 technical papers (about 150 of these in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. He has written (jointly with C. M. Krishna) a textbook "Real-Time Systems" which is scheduled to be published by McGraw Hill in 1996. In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award for a paper on robot trajectory planning. In 1989, he also received the Research Excellence Award from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing.

He has also been applying the basic research results of real-time computing to multimedia systems, intelligent transportation systems, and manufacturing applications ranging from the control of robots and machine tools to the development of open architectures for manufacturing equipment and processes. (The latter is being pursued as a key thrust area of the newly-established NSF Engineering Research Center on Reconfigurable Machining Systems.)

He received the B.S. degree in Electronics Engineering from Seoul National University, Seoul, Korea in 1970, and both the M.S. and Ph.D degrees in Electrical Engineering from Cornell University, Ithaca, New York in 1976 and 1978, respectively. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division within the Department of Electrical Engineering and Computer Science at UC Berkeley, and International Computer Science Institute, Berkeley, CA, IBM T. J. Watson Research Center, and Software Engineering Institute at Carnegie Mellon University. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning January 1991.

He is an IEEE fellow, was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, the Guest Editor of the 1987 August special issue of *IEEE Transactions on Computers* on Real-Time Systems, a Program Co-Chair for the 1992 *International Conference on Parallel Processing*, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-93, was a Distinguished Visitor of the Computer Society of the IEEE, an Editor of *IEEE Trans. on Parallel and Distributed Computing*, and an Area Editor of *International Journal of Time-Critical Computing Systems*.