OODB Support for Real-Time Open-Architecture Controllers

Lei Zhou, Elke A. Rundensteiner, and Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{lzhou, rundenst, kgshin}@eecs.umich.edu

Abstract

Open-architecture machine tool controllers have been an important research subject in both academia and industry. They are hard real-time applications that require a built-in database management system to support concurrent data access and provide well-defined interfaces. These advanced applications often utilize object-oriented models to handle complex data types. Since there exists no agreed-upon real-time object model, we 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 control applications, namely, timing constraints and performance polymorphism. It uses specialization dimensions to model timing specifications and letter class hierarchies to capture performance polymorphism. Based on the model, we develop a prototype controller on our open-architecture machine tool controller testbed. The controller has been successfully used for a 6-axis robotic table.

1 INTRODUCTION

Machine tool controllers have become more sophisticated in recent years by capitalizing on the technological progress made in the field of computer. However, problems still remain in terms of life-cycle cost and lack of openness in commercially available controllers. There has been considerable interest on the subject, in both academia and industry, North America and Europe. Two primary examples of this activity are the OSACA project [19] in Germany and the Enhanced Machine Controller (EMC) project [1] at the National Institute of Standards & Technology. There is a general consensus that the controller 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. The VME processor board is an example of a hardware module, while a POSIX-compliant operating system kernel 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.

Modular machine tool controllers require a built-in database management system (DBMS) to support concurrent data access and provide well-defined interfaces between different software entities (tasks, processes, and modules). They typically are subject to a range of timing constraints, which require the DBMS to provide timing guarantees, sometimes under complex conditions. The deadlines of real-time tasks can be classified as *hard*, *firm*, or *soft* [21]. A deadline is hard if the consequences of not meeting it can be catastrophic, such as 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 consequences of not meeting the deadline are not catastrophic, e.g., weather forecast. A deadline which is neither hard nor firm is soft. The utility of results produced by a task with a soft deadline decreases over time after the deadline expires. Conventional DBMSs generally have no mechanisms to specify, and much less to enforce, such complex timing guarantees. Thus, they do not offer the performance levels or response-time guarantees needed by real-time applications.

Such inadequacy has recently spawned the field of real-time databases (RTDBs) [11, 18, 21, 22, 24]. Most of RTDB research has been focused on soft real-time constraints, and many transaction management algorithms have been proposed for this purpose. We instead offer a new paradigm for hard RTDB applications, especially machine tool controllers. Figure 1 demonstrates the role of a RTDB in machine tool controllers: it provides data access services and well-defined interfaces for data sharing among different modules within a controller and/or among cooperating controllers, and facilitates module reuse across controllers (the shaded module is used by both controllers). The RTDB may maintain actual reusable modules or just references to them.

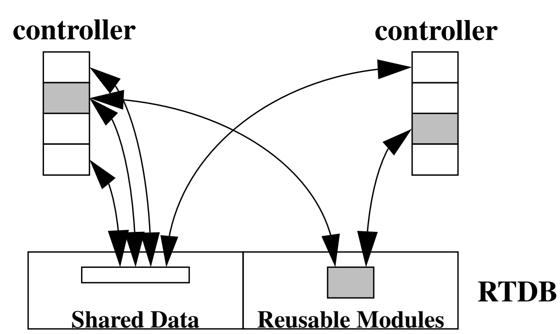


Figure 1. Data Sharing and Module Reusing of Controllers

The object-oriented approach has been shown to be effective in managing the development and maintenance of large complex systems, including real-time systems [5, 7]. It is the

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

choice of technology for controllers as well. Although real-time database research often uses the object-oriented paradigm, no agreed-upon real-time object-oriented data model is available at this time. Therefore, we define in this paper a real-time data model that is suitable for manufacturing applications. We have evaluated existing models used for real-time applications [3, 9, 12, 13, 15, 17, 25]. Based on this evaluation and the characteristics of manufacturing applications, we extract a simple yet powerful real-time object model. It explicitly captures important characteristics of RTDB applications, especially in the manufacturing application domain, namely, timing constraints and performance polymorphism. It uses specialization dimensions to model timing specifications and letter class hierarchies to capture performance polymorphism. Although regular object-oriented programming techniques (e.g., composite object classes) may be used to implement the proposed concepts, they neither explicitly capture these concepts nor provide a mechanism to enforce them.

The remainder of the paper is organized as follows. Section 2 describes a conceptual real-time object model, while Section 3 presents the application of the model to a motion controller. Section 4 briefly covers related work, and the paper concludes with Section 5.

2 CONCEPTUAL REAL-TIME OBJECT MODEL

Machine tool controllers are hard real-time applications. All operations, from reading sensor data to issuing actuator commands, must be completed within each control period. Timing requirements are intrinsic to these operations. Any failure to meet timing constraints may cause severe property damages and human injuries. At the same time, the openness of the controller clears the path to the general availability of compatible modular software (and hardware) components, to be offered by different vendors. An application-programming interface (API) that automates this selection process based on application requirements would be a major help for application developers. This is the main goal of our real-time model design, namely, to provide facilities for simplifying reuse of modules, for increasing the productivity of real-time application developers, and for keeping application-code as constraint-optimized as possible given an up-to-date library of application-specific kernel classes. In this section, we describe the *Real-time Object Model with Performance Polymorphism* (ROMPP). It is *conceptual* in the sense that it is not dependent on any specific implementation. This model aims to provide a simple, yet sufficiently powerful foundation, for our real-time data management research for open-architecture machine tool controllers by explicitly capturing key application characteristics. In other words, we are not proposing a *complete* data model, instead, one that is suitable for manufacturing applications.

2.1 Object-Oriented Concepts

Machine tool controllers have become more sophisticated in recent years by capitalizing on the progress of computer technology. However, problems still remain in terms of life-cycle cost and lack of openness in commercially available controllers. The object-oriented approach has been shown to be effective in managing the development and maintenance of controllers. Therefore, our data model ROMPP is object-oriented, that is, any real-world entity is represented by an *object*. ROMPP adopts basic object-oriented concepts, such as class and inheritance, as can be found in most object-oriented models [6, 10, 14]. 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 as value either a system-provided object, such as an integer, or a user-defined object, such as a sensor. Instance variables are* **private** *to the object, i.e., they can only be accessed by the object's methods. A method is defined by (***signature***,* **body***), where the signature consists of a method name M and a mapping from input parameter specifications to an output parameter specification:* $\mathrm{M}\ (\mathrm{In}_1, \mathrm{In}_2, \ldots, \mathrm{In}_n) \rightarrow \mathrm{Out}$. *A parameter specification (either input or output) is a class name. The body corresponds to the actual code which implements the desired functionality of the method. Methods can be either private or* **public***. A public method is accessible to all methods of the object or even to other objects. An instance variable V_i of an object A can be specified as being* **composite***. In this case, the object B referenced through the composite instance variable V_i is owned by the object A. Deletion of A will cause the deletion of B.*

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

Definition 3. For two classes C_1 and C_2, C_1 is a **subclass**[1] *of C_2, denoted C_1* **is-a** *C_2, if and only if C_1 inherits every instance variable and method of C_2.*

Multiple inheritance is allowed, that is, a class can have more than one superclass. Note that private instance variables and methods of a class are not visible to its subclasses, although they are inherited by the subclasses. Only public methods of the superclasses are accessible to the subclass and become part of its public interface. In other words, private

1. Throughout this paper, we say that A is a *subclass* of B (B is a *superclass* of A) iff A inherits directly from B, and A is a *descendant* of B (B is an *ancestor* of A) iff A inherits directly/indirectly from B.

instance variables inherited from a superclass are stored in the instances of the subclass, but these private instance variables (and methods) can only be accessed by the subclass via public methods defined in the superclass. A public method of a class can be declared virtual, i.e., it has no code associated with it and must be implemented in the class' subclasses (or descendants). The objects of the same class type are usually called *instances* of the class.

Definition 4. A **class hierarchy** *is a directed acyclic graph*[2] *(DAG) $S = (V, E)$, where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on $V \times V$ that represents all subclass relationships between all pairs of classes in V. In particular, each directed edge e from C_1 to C_2, denoted by $e = \langle C_1, C_2 \rangle$, represents the is-a relationship (C_1 is-a C_2).*

Definition 5. A **schema** *is the description of a database. An OODB schema is equal to the class hierarchy.*

2.2 Key Characteristics

Based on our evaluation of existing real-time systems [3, 9, 12, 13, 15, 17, 25] and real-time manufacturing applications [2, 4, 16], we have identified two key characteristics for real-time data models: *timing constraints* and *performance polymorphism*. In manufacturing automation applications (machine tool controllers, in particular), control tasks periodically read sensor data, compute control parameters, and issue actuator commands. All these operations must be completed within each control cycle; otherwise, it may cause catastrophic consequences. Timing constraints are an essential characteristic of such applications. 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, which we call performance polymorphism, is 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 automation application developers.

2.2.1 Timing Constraints

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

Definition 6. The **timing constraint** *of a task refers to the deadline by which the task must be completed.*

In our real-time object model, timing constraints are associated with the performance of methods, since the behavior of an object is represented by its methods. Applications will be requesting services from objects via their respective methods. We thus need to extend the definition of a method (Definition 1).

Definition 7. A **method** *in ROMPP is now extended to be defined as a triple (***signature***,* **body***,* **performance***), with signature and body defined as in Definition 1. The (optional) third field specifies the performance measure of the method, such as execution time, memory space, etc.*

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

2.2.2 Performance Polymorphism

To implement the functionality of a method, typically several different algorithms and/or data structures can be used. Machine tool controllers need support in selecting one from these implementations based on performance and/or price, optimizing the objectives of the control applications. For example, a controller that controls a milling machine may choose among a variety of control algorithms, such as adaptive control, linear and nonlinear control algorithms. The controller may want to select among these different control algorithms based on performance characteristics, but without having to deal with details of the respective implementation. Such selection may occur either at application start-up time or at run time. This second key characteristic of a real-time model is called performance polymorphism.

Definition 8. **Performance polymorphism** *refers to the concept of maintaining and selecting among multiple implementations of a method (body) that carry out the same task and differ only in their performance measures, such as execution time, memory space, system configuration, result precision, and so on. Performance polymorphism is explicitly supported by ROMPP, allowing dynamic selection of the most appropriate method implementation by the system based on performance characteristics desired by the application.*

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

1. The knowledge of performance polymorphism is captured and maintained separately from the schema. For example, the service designer[3] may use a library to group different implementations of the same service. The knowledge about such real-time object libraries is not part of the system schema. Although the schema may include a description of different implementations of the service, it provides no help to the application developer on how to

2. A class hierarchy without multiple inheritance corresponds to a tree rather than a DAG.

3. 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.

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. Obviously, such approaches do not provide support for software reusability, and put all burden on the application developer.

2. The service designer could use one implementation of an object to meet all performance requirements, no matter how different they are. This over-simplified approach would typically require us to assume a worst-case scenario. This is not even always possible, because requirements may contradict one another. It also wastes resources and poses true limitation on applications. For example, suppose the system has a memory space of 10MB, and the chosen implementation of object A requires 8MB while object B needs 3MB. Obviously, A and B cannot co-exist in memory. Therefore, a real-time task cannot receive services from A and B concurrently, even if A needs only 2MB to provide the desired services for this particular application when using a slightly slower algorithm.
3. Another option is to duplicate the definition of the method (or object) with each of its implementations and give them distinct names in order to simulate performance polymorphism. This would again carry all disadvantages of the first approach above, making the application developer responsible for maintaining information about individual services and their relationships. In addition, a system of such a type is difficult to maintain. Any change in the definition of the method has to be made to all its duplicates, which is inefficient and often prone to errors.

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

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

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

Definition 9. An **envelope/letter structure** *is a composite object structure formed by a pair of classes that act as one: an outer class (***envelope class***, or* **EC***) that is the visible part to the user, and an inner class (***letter class***, or* **LC***) that buries implementation details.*

Definition 10. A **letter class hierarchy** *is a class hierarchy as defined in Definition 4 that consists of, as its root, an envelope class and zero to many letter classes. The envelope class and all its letter classes must have exactly the same public methods. Furthermore, the letter classes can only have is-a relationships with classes in the same letter class hierarchy. Letter classes are not explicitly accessed by the application developer, but rather manipulated by the system based on the performance requirements specified with the envelope class.*

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

Definition 11. An **envelope class hierarchy** *is a class hierarchy that consists of, as its root, a system-provided class, called* **ROOT***, and one or more envelope classes.*

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

Definition 12. A **specialization dimension** *is a performance measure (Definition 7) that distinguishes letter classes from one another. A specialization dimension must be assigned to a public method in the letter class hierarchy. There is a* **specialization space** *associated with each letter class hierarchy and its axes are specialization dimensions.*

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

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

2.3 Model Constructs

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

1. @EC <ec>

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

2. @LC <lc> OF <ec>

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

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

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

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

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

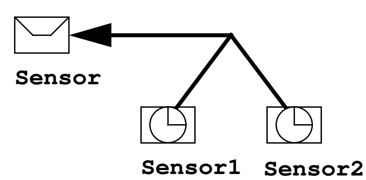
An example is given below to illustrate the newly introduced concepts. It is described in C++, since C++ and C are among the most popular programming languages for real-time applications. By placing the model constructs in programming language comments, we avoid modifying the programming language itself.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

Example: A Letter Class Hierarchy with Two Specialization Dimensions

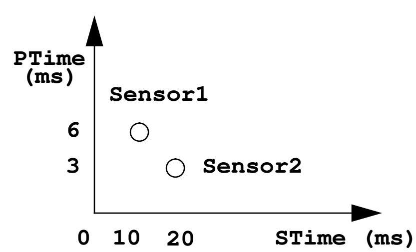
In Figure 2, the class **Sensor** is the envelope class, while classes **Sensor1** and **Sensor2** are its letter classes. There are two specialization dimensions, associated with the methods **sample()** and **process()** (identified as **STime** and **PTime**), respectively. Therefore, the specialization space is a plane, as shown in Figure 2(c). Note that specialization dimensions may not necessarily be inferred from the structure of the letter class hierarchies as, for instance, shown in Figure 2(b), since these simply capture is-a relationships in terms of property inheritance.

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

(a) Model Description



(b) Letter Class Hierarchy



(c) Specialization Space

Figure 2. Example of Two-Dimensional Specialization Space

2.4 Real-Time Object-Oriented Database Schema

Definition 13. A **real-time object-oriented database (RTOODB) schema** *is composed of one envelope class* *hierarchy and a set of zero or more letter class hierarchies, defined in Definition 11 and Definition 10, respectively. Each letter class hierarchy is associated with one envelope class.*

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

2.5 Application-Programming Interface (API)

ROMPP offers a unique application-programming interface (API) for manufacturing applications. Because ROMPP explicitly models timing constraints and performance polymorphism, it naturally supports an automated mechanism in the API that selects software modules based on requirements of the application. This mechanism is visualized in Figure 3. For example, the service designer provides a collection of system services that constitute the kernel of the RTOODB. When a machine tool controller (being built by the application developer) needs some service, it sends a service request, which specifies the type of service, performance constraints, and other requirements, to the API. The mechanism in API, enabled and explicitly supported by 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. We will show the application of the model to a real working application in Section 3.

Example: A RTOODB Schema

Figure 4 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 following constraints:

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

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

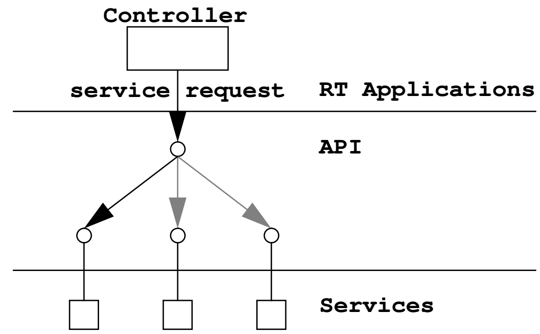


Figure 3. Application-Programming Interface for Manufacturing

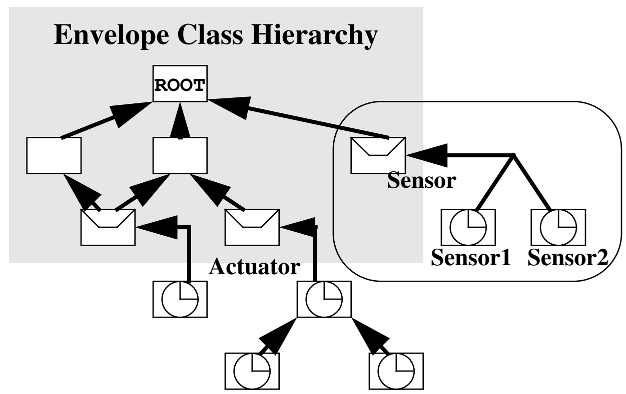


Figure 4. Example Real-Time Object-Oriented Database Schema

3 APPLICATION OF MODEL TO A PROTOTYPE CONTROLLER

We now want to demonstrate the utility of the real-time object model ROMPP defined in Section 2. In particular, we want to discuss how it can be used to build real-time machine tool controllers. Figure 5 depicts our open-architecture machine tool controller testbed. Control tasks are executed on VME-based processors boards (e.g., Motorola 680x0s and Intel 80x86s) running a real-time operating system (e.g., VxWorks or QNX), in order to achieve good performance and timing predictability. 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., Controller Area Network or CAN, and SERCOS). Control software may be cross-developed on and downloaded from remote workstations 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 supply a foundation of software openness.

The key concepts of ROMPP, namely, using specialization dimensions to characterize timing constraints and using letter class hierarchies to capture performance polymorphism, are incorporated in MDARTS [15]. MDARTS is a multiprocessor database architecture for real-time systems, built in C++ at the University of Michigan. To evaluate the suitability of the MDARTS in the domain of real-time manufacturing control applications, a prototype motion controller for a six degree-of-freedom (DOF) robotic manipulator was implemented (Figure 6). It is a physical mechanism for geometric error compensation at the assembly stage of automotive applications. This mechanism includes a multi-axis manipulating device (essentially a robotic table to which sheet metal parts can be affixed), and a multi-axis servo-motion controller that handles the execution of desired motions at the manipulator joint level. The servo-motion controller board is a Programmable Multi-Axis Controller (PMAC) designed and manufactured by Delta Tau Systems. The manipulator consists of a fixed base, a movable platform, and six independently positioned legs. Each leg is connected to the base by a 2-DOF joint on one end, and to the platform by a 3-DOF joint on the other end. The tops of adjacent legs are joined together at the platform connection point, forming a set of three leg triangles.

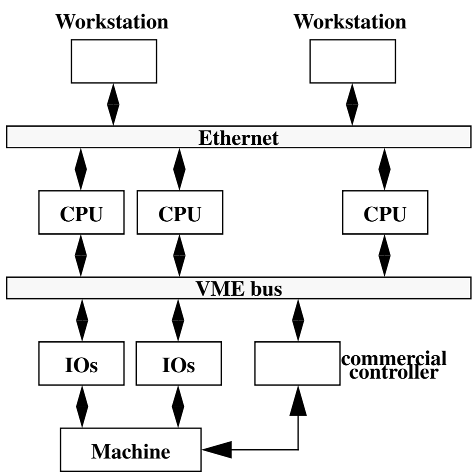


Figure 5. Open-Architecture Controller Testbed

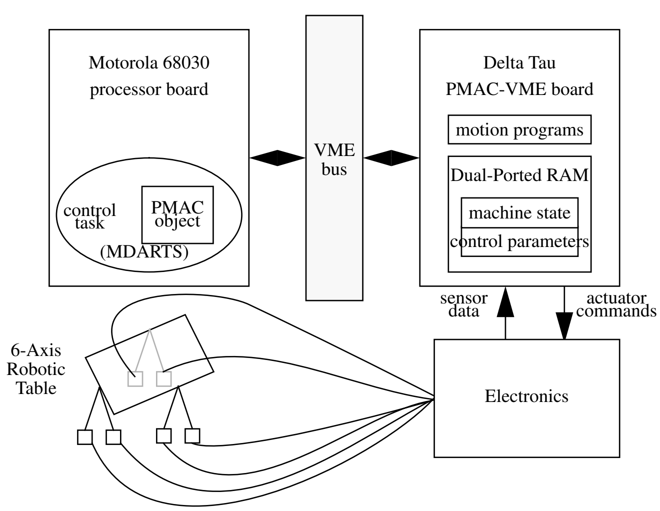


Figure 6. MDARTS Experiment Setup (using a ROMPP model).

The PMAC board executes downloaded motion programs to manipulate the robotic table. There is a dual-ported RAM on the PMAC board, where machine state and control parameters are stored. The PMAC is encapsulated in the **Pmac** object of the control task. During each control cycle, the control task reads machine state, computes control parameters, and sends them to the PMAC. The PMAC then adjusts the motion trajectory of the robotic table based on the control parameters and the original motion program, and updates machine state. Timing constraints for data access (which restrict the methods **Pmac::getValue()** and **Pmac::setValue()**) and computations are specified for the **Pmac** object in order to ensure that all operations can be finished within each (periodic) control cycle.

There are several system-defined specialization dimensions in MDARTS: read time, write time, priority, persistency, etc. Constraints can be specified for these pre-defined and user-defined specialization dimensions. The most appropriate data access services are selected automatically at the application start-up time based on the above specified timing requirements, therefore, reducing run-time overhead to a minimum. This method of binding at initialization time is very suitable for meeting the needs of manufacturing automation applications, since timing information may be obtained beforehand. In fact, this is the only way that the granularity of performance (as small as tens of microseconds) required by the application can be achieved.

Furthermore, 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 manually. With the automated mechanism, all the application developer needs to do is change the requirement specifications (in the case of application requirement changes) or nothing (in the case of service implementation changes). The system then will take care of the service selection and binding. It was shown that this prototype controller is able to monitor and modify the path of the manipulator while it is executing a sequence of move commands. This experiment thus demonstrates, among other features of MDARTS, that our real-time object model is useful in practice.

4 RELATED WORK

While a large body of work on real-time systems exists, no agreed-upon, conceptual model for real-time databases has been established. In this paper, we show that timing constraints and performance polymorphism are two key characteristics for real-time manufacturing applications and should be explicitly supported by a real-time data model. Unlike our work, none of the existing models were specifically targeted to real-time open-architecture controllers. Due to limited space, interested reader is referred to [26] for a comparison between ROMMP and existing real-time models, such as CHAOS [3, 20], ARTS [17, 23], RTC++ [12], Flex [13], HiPAC [9], RTSORAC [25], and MDARTS [15].

5 CONCLUSIONS

In the paper, we identified timing constraints and performance polymorphism as two key characteristics of real-time manufacturing applications. We then presented a conceptual real-time object model, ROMPP, which provides a simple, yet sufficiently powerful foundation for our real-time data management research for open-architecture machine tool controllers by explicitly capturing these key characteristics. In other words, we have not proposed a *complete* real-time data model, instead, described one that is suitable for manufacturing applications. Our real-time model provides facilities for optimized reuse of modules, for increasing the productivity of real-time application developers, and for keeping application-code as optimized as possible given a collection of application-specific kernel classes in ROMPP. We also demonstrated the applicability and usefulness of the proposed concepts for real-time manufacturing applications, in particular, open-architecture machine tool controllers. We discussed how this model was successfully used in building a real-time machine tool controller, namely, a prototype motion controller for a six degree-of-freedom robotic manipulator.

As next step, we are investigating the impact of schema evolution technology on real-time OODBs in terms of reducing turn-around time for developing real-time control applications, and for reusing the most appropriate kernel classes with minimal effort. Preliminary results are reported in [26].

6 ACKNOWLEDGEMENTS

This research was supported in part by the Horace H. Rackham School of Graduate Studies at the University of Michigan under a Research Partnership Grant, the United Parcel Service Foundation under an IVHS Graduate Fellowship, and the National Science Foundation under Grants DDM-9313222 and IRI-9309076.

7 REFERENCES

[1] James Albus, presentation at the *Int'l Workshop on Open-Architecture Controllers for Automation*, Ann Arbor, MI, Apr. 1994.

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

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

[4] Sushil Birla, "Conceptual Modeling of Manufacturing Automation," *CSE-TR-220-94*, The University of Michigan, Oct. 1994.

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

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

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

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

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

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

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

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

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

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

[15] Victor B. Lortz, "An Object-Oriented Real-Time Database System for Multiprocessors," *Ph.D. dissertation*, The University of Michigan, March 1994.

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

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

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

[19] Günter Prischow and Gerd Junghans, presentations at the *International Workshop on Open-Architecture Controllers for Automation*, Ann Arbor, Michigan, April 1994.

[20] Karsten Schwan, et al., "CHAOS-Kernel Support for Objects in the Real-Time Domain," *IEEE Transactions on Computers*, Vol. C-36, No. 8, Aug. 1987, pages 904-916.

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

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

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

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

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

[26] Lei Zhou, Elke Rundensteiner, and Kang Shin, "Schema Evolution for Real-Time Object-Oriented Databases," *CSE-TR-199-94*, The University of Michigan, March 1994.