Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

High-Level Reuse in the Design of an Object-Oriented Real-Time System Framework

Win-Bin See[†] and *Sao-Jie Chen*

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.
email: csj@cc.ee.ntu.edu.tw

Abstract

Reuse is one of the major strategies to increase software productivity. Object-oriented techniques provide sound mechanism to support the possibility of reuse. Yet, in order to promote the effectiveness and scale of reuse, augmenting the level of reuse is a promising direction to endeavor. In the implementation of our Object-Oriented Real-Time System Framework (OORTSF), we have attempted to design a reusable object-oriented software framework for embedded real-time applications. Later, with the proliferation of design patterns, we found the concept of design patterns be helpful in the documentation and assessment of our design. Besides, from the aspect of promoting the applicability of our framework, we found that through adapting our framework for some open software architecture, like CORBA, we can setup a path to extend the reusability of our OORTSF towards distributed systems. In our experience, design pattern, framework, and architecture complement each other and will become a synergy of object-oriented technologies.

Key Words: Object-Oriented Paradigm, Software Reuse

1. Introduction

Software development cost has always been a concern in software industry. Software reuse is a direction towards software development cost reduction, and augmentation in the level of reuse entities will further strengthen the feasibility of reuse to a larger scale. Recently, three high level software reuse technologies, (1) Design patterns [1,2], (2) Frameworks [3,4], and (3) Software architectures [5,6], are evolving rapidly and some exhilarant results have been reported [7].

In what follows, we discuss the high-level reuse techniques based on: design patterns, frameworks, and software architectures, and describe the design of our object-oriented real-time system framework (OORTSF). Then, we show how to use design patterns to document and assess the OORTSF, and how to extend OORTSF to an open architecture platform. Finally, a conclusion is given.

2. High Level Reuse Techniques: Design Patterns, Frameworks, and Software Architectures

Using terms similar to those have been used in parallel processing, software reuse can be classified as (1) low-level, or fine-grain reuse, that reuses only small amount of code segments at programming level, (2) high-level, or coarse-grain reuse, that reuses design concepts and/or larger amount of codes. Promoting the level of reuse is important in increasing the scale of software reuse. In this article, we will focus on high-level reuse. Promising technologies in high-level reuse are: (1) Design Patterns, (2) Frameworks, and (3) Software Architectures. Figure 1 shows a high-level reuse technology hierarchy.

- **Design Patterns** provide solution to a general design problem using sets of related classes and objects. The description of design patterns often provides metaphorical abstractions to make users capture the concept easily and to use the patterns effectively. Typical examples of design patterns are, (1) structural pattern "Facade," which intends to provide a unified interface to a set of interactions in a subsystem; (2) "Observer," which defines a one-to-many relationship to reflect the changes of an

[†] Win-Bin See works for Aero Industrial Development Company, Taichung, Taiwan, R.O.C.

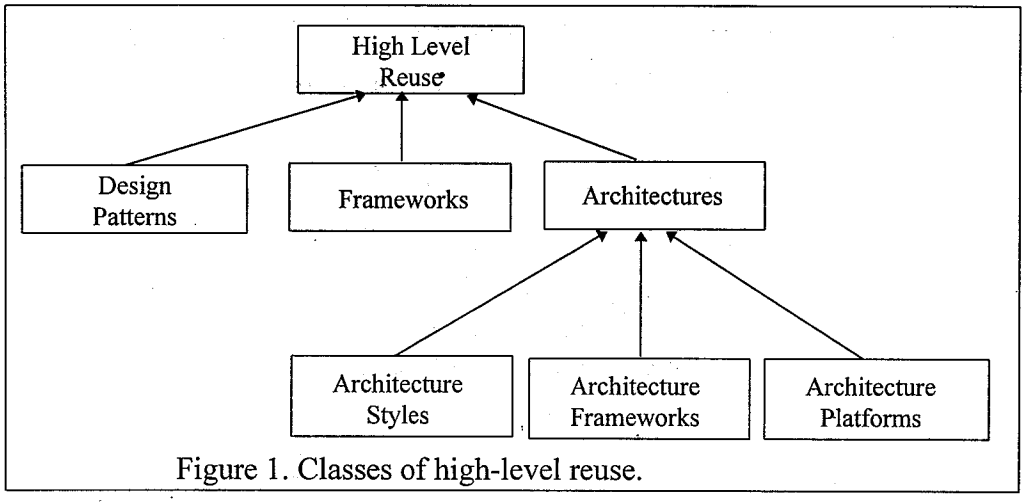

Figure 1. Classes of high-level reuse.

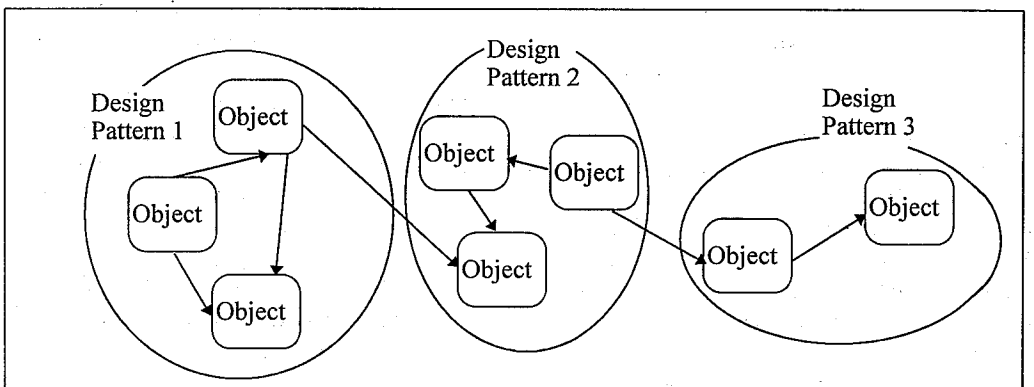

Figure 2. Using design patterns to construct an application.

object state into different forms of view [1]. Design patterns have been used as a tool to document a larger reusable software entity, the framework [8]. Design patterns can also be used to design or to help in assessing the design of an application [1,2]. Figure 2 shows that application software can be thought of or actually composed of design patterns. The content of a design pattern usually provides rich semantic descriptions. Hence, a common set of design patterns can enrich the communication among different parties joining a software development process.

- **Frameworks** provide a set of collaborating classes and run-time objects to facilitate the creation of software in some specific application domain. Figure 3 shows typical usage of a framework, a specialization of underlying framework to create application specific software. Figure 4 shows a framework for multimedia applications [4]. In this framework, domain object classes were identified and organized. Media format class is used to abstract various media formats, such as JPEG image data or ASCII text format data. Media component class encapsulates the physical media component, such as video mixer and image scanner.

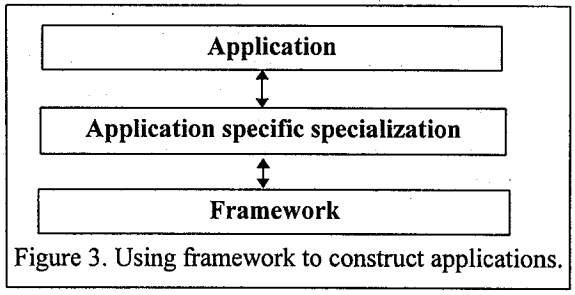

Figure 3. Using framework to construct applications.

- **Software Architectures**

Crafting software is also called to architect a software. An operational software has some sort of architecture inside. Since we are interested in the reuse aspect of software architectures, we further differentiate the reusable entities in software architectures into three different categories, (1) architecture style, (2) architecture framework, and (3) architecture platform. The differences among them and examples for each category are described as follows.

- A software *architecture style* shows

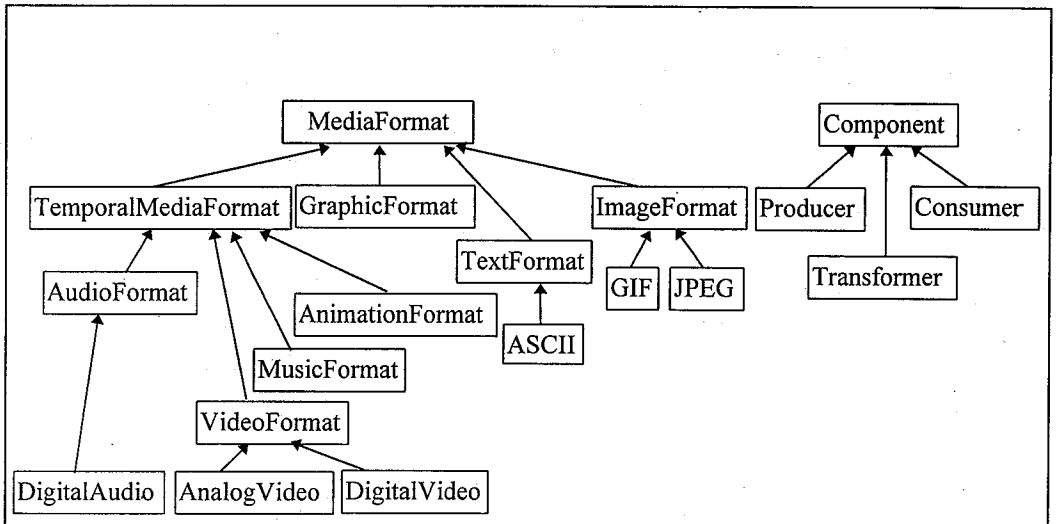

Figure 4. Multimedia Framework.

a well-established common form of global software organization. For example, according Shaw and Garlan [6], seven styles of architectures have been introduced, they are *Pipes and Filters*, *Object-oriented organizations*, *Implicit invocations*, *Layered systems*, *Repositories*, *Interpreters*, and *Process control*.

- An *architecture framework* is a more detailed and complete framework using one or more architecture design styles for some specific domain application development. A software architecture artifact designed with some style can be reused as an architecture framework, if only it provides enough documentation and builds enough flexibility inside to encompass a certain application domain.

  In the article [9], an architecture for telecommunication-infrastructure systems has been introduced. Since it is an architecture with more specific domain of application, we categorize it as an architecture framework. Although, this architecture has been claimed to be based on "building blocks" instead of being "object-oriented", we believe that it is possible to have "object-oriented" counterpart for this kind of system.
- An *architecture platform* provides a flexible infra-structure to fit a wide range of applications. An architecture platform is similar to an operating system in providing some basic services to the application software developed on it. Yet, architecture platforms are designed to provide cross applications software communication. Thus, they can be served as an infra-structure to promote the object level collaboration and reuse, The Common Object Request Broker Architecture (CORBA) and Specification of Object Management Group (OMG) [5,10] is an example of this. Figure 5 shows a layered architecture view of the OMG CORBA reference model. Under CORBA, all communications between components are managed by the ORB (Object Request Broker). The ORB is the foundation of OMG solution in an open distributed computing environment. CORBA supports applications to run in heterogeneous computing platforms, insulates applications from the variations in hardware platforms and in operating systems.

Table 1 summarizes the distinctive attributes of these high-level reuses. We compare different high-level reuse categories with the following

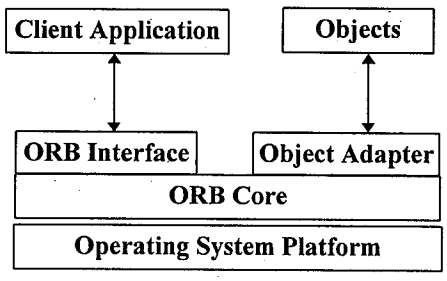

Figure 5. An architectural view of the OMG CORBA model.

Table 1. High level reuse styles and related features

Attributes \ Reuse Styles	Design Patterns	Frameworks	Architectures
Examples	Gamma et al. [1]	Multimedia Framework [4]	Architecture Styles [6]; Telecom. architecture [9]; OMG CORBA [5,10]
Reuse entities	Conceptual, Design, Code	Design, Code	Conceptual; Design, Code; Object
Application domains	Wide range	Specific	Wide; Specific; Wide
Coupling with computing platforms	Low	High	Low; High; Low
Effort (provider)	Low	Medium to High	High
Effort (user)	Medium	Medium	High

attributes: (1) reuse entities, (2) application domains, (3) coupling with computing platforms, and (4) the amount of the effort spent by providers and users of the reuse items.

- In first row of Table 1, examples for three different high-level reuses are given. In the last column of the examples row, three items are listed, each item corresponds to one of the three sub-categories of architecture-level reuse.
- In the row of reuse entities, we identify four reusable entities: (1) conceptual entity, (2) design, (3) code, and (4) object. Reuse in object entity means to reuse the run-time object, the component software concept [5] belongs to this category, CORBA is an open standard architecture platform which can be used to develop component software.
- In the row of application domains, design patterns are solutions to general problems, thus it has a wide range of application domains. A framework will be more specific in its domain of application. In the architecture-level of reuse, an architecture style can be used in different application domains, while an architecture framework will be in a more specific application domain.
- In the row of coupling with computing platforms, the framework and architecture framework will be more specifically developed, hence the coupling of them with computing platforms will be higher, the reuse of these entities to different platforms need to consider more about inter-platform porting issue.
- As shown in the effort rows in the table, the provider and the user have to spend high effort in producing the reusable architecture artifact and customizing it, respectively; but the scale, and hence the potential benefit, of reuse in this level will also be prominent.

3. Object-Oriented Real-time System Framework (OORTSF)

Real-time systems must be suitably constrained in order to be able to verify the timing behavior. One way to constrain a real-time system to have predicted timing is to implement it as a system of periodic tasks. For periodic tasks, three real-time task scheduling methods have been proven to be theoretically sound and practically useful, they are: (1) cyclic scheduler (CYS), (2) rate monotonic scheduler (RMS), and (3) earliest deadline first scheduler (EDF) [11]. In our framework, the above three scheduling methods are supported. Users can choose whichever is suitable for their need.

Object-oriented paradigm is pledged for (1) its power in modeling real world objects, and (2) its reusability. Providing an object-oriented real-time system framework will help the real-time application system developers to integrate their real world objects into this framework more naturally. Hence, we design a framework using object-oriented paradigm, called the object-oriented real-time system framework (OORTSF), where abstract class concept of object-oriented paradigm is used to extract common behavior of the objects under consideration. For example, we defined an abstract class `CSchedulerBase` to capture the basic behaviors of the schedulers needed; then an inheritance mechanism is used to specialize this abstract base class into three schedulers differentiated by their scheduling methods: (1) `CCYScheduler`, (2)

CRMScheduler, and (3) CEDFScheduler. Figure 6 shows the class hierarchy diagram of scheduler-related classes and their associated attributes and member functions in OORTSF.

System supporting classes other than scheduler and application tasks in OORTSF, such as: (1) TimeBase, (2) Exception Handler, (3) Fault Manager, (4) Shared Resources, (4) I/O Data Link, and (5) TMR (Triple-Module-Redundancy) Interface, are presented as follows.

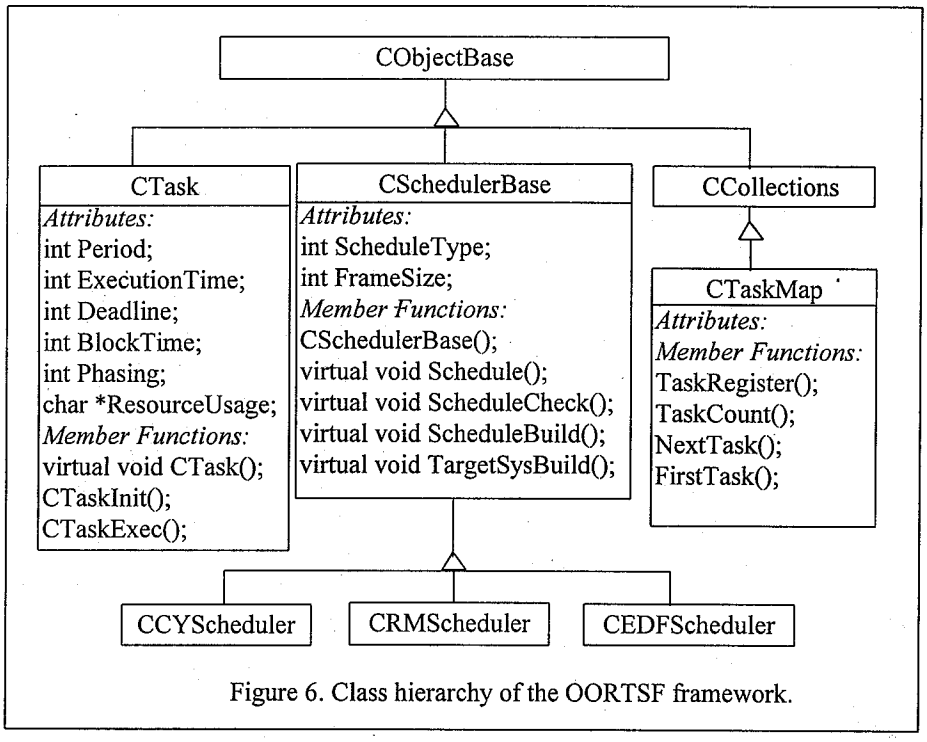

Figure 6. Class hierarchy of the OORTSF framework.

- `CTimeBase`
  This is the time base class. A system time is provided by an object of this class. The system time resolution can be adjusted by refining the behavior of this class. The execution time frame size is also determined in this class by interrupting the system at periodic intervals.
- `CExceptionHandler`
  Any exception condition, e.g., system power failure and machine errors, will cause a system exception. After exception handler performs the system status saving, it will invoke the fault manager for further handling and fault reporting.
- `CFaultManager`
  In a typical embedded application environment, the memory tends to be of limited size. To reduce the memory consumption, we give a fixed-sized table to manage fault collection. Figure 7 shows the definition of `CFaultManager` class. The information kept inside the `fault_list` of the `FaultManager` object contains: fault identification code (`fault_id`), fault occurrence time (`fault_time`) which shows the first occurrence of the fault, fault severity level (`fault_severity`), fault occurrence count (`fault_cnt`), and fault message. The occurrence count of faults will be helpful for post mission trouble shooting.
- `CSharedResources`
  This class provides a semaphore operations to guard shared resources in the application to ensure serialized access to it. The definition of this class is shown in Figure 7.
- `CI/ODataLink`
  This class provides the basic input and output function of the system.
- `CTMRInterface`
  As a variant of our kernel design, a TMRInterface is introduced. It is because our project requires the kernel to support system with fault tolerant hardware. This class is responsible for the interfacing with a TMR design.

A real-time application using OORTSF consists of a scheduler object, a task-map object, and one or more application task objects, and objects of the above-mentioned supporting classes. In Figure 6, a variable started with a leading "C" means that it is a class definition. Directed arrow shows the inheritance relation between classes, the arrow points from derived class to superclass. Each application task

definition must be encapsulated in a class derived from `CTask`. The scheduling related parameters of the task, needed by the scheduler, are defined as attributes of this class and they should be provided by the system user. A task-map object, instance of `CTaskMap`, is used to collect the application task objects. The collection operation is done via the `TaskRegister` method of the task-map object. A scheduler object is an instance of one of the three scheduling-algorithm-specific scheduler classes: `CCYScheduler`, `CRMScheduler`, and `CEDFScheduler`. All these three classes are derived from `CSchedulerBase`. This framework can be operated in two phases: (1) Schedulability check phase, (2) Application execution phase. In the schedulability check phase, all application tasks will have to be registered into a task-map object via the `TaskRegister` method, the task-map object will be analyzed by the `ScheduleCheck` method of the scheduler object to check the schedulability. If the application tasks get through the schedulability check successfully, the tasks can be scheduled for normal system operations, that is to enter phase two operation. After the success of schedulability check, user has an option to generate a compact target system containing only execution codes of application tasks and framework kernel. This job is done via the `TargetSysBuild` method of the scheduler object. Another way to enter phase two operation is using the `Schedule` method of the scheduler object to start the application directly. Figure 8 shows the schedulability check and target system build operations of this framework. We have also devised an interactive user friendly graphical user interface for the above operations which includes task parameters entry, schedulability check reports, and application task build [12]. Figure 9 shows a set of typical collaborating objects in an embedded system using OORTSF.

4. Design Patterns Assessments of OORTSF

Users of a framework do not need to have thorough understanding about the design details of the framework. Using design pattern as a description language to document a framework, a framework user will be able to have a quick understanding about the mechanisms used in the framework, specially when the user already has general knowledge about the design patterns used in the document. In what follows, we discuss some design patterns which can be used to document and assess the OORTSF. Gamma categorized the design patterns [1] based on their purpose into (1) creational patterns, (2) structural patterns and, (3) behavioral patterns. For the assessment of OORTSF, we choose three structural patterns: Adapter, Bridge, and Proxy; and three behavioral patterns: Iterator, Observer, and Strategy.

- Adapter: It converts the interface of a class into another interface expected by a client. Adapter lets classes work together that could not otherwise because of incompatible interface. In OORTSF framework, application tasks need to be converted to the format of the `Ctask` class, Adapter pattern can be used by the users of the framework to adapt their tasks to the framework.
- Bridge: This is a structural pattern to separate the abstraction from its implementation. Since we have cropped three different kinds of real-time scheduling methods in OORTSF and there is possibility to introduce some other scheduling methods, it looks feasible to use the Bridge pattern for the design of a scheduler class. However, in order not to introduce overhead of using pointers in Bridge pattern design we do not adopt the Bridge pattern in our scheduler class design.
- Proxy: This pattern provides a surrogate or placeholder for another object to control access to it. After the scheduler schedules some application task objects to execution according to the scheduling method used, the scheduled tasks have to be dispatched to the processor. Proxy pattern can be used to hold the entry point of an application task for dispatching.
- Iterator: Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. In OORTSF, the schedulability check can be implemented in a task-map object to access the required task during the schedulability check phase and the application execution phase.
- Observer: This is a behavioral pattern to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This pattern is used to implement the phase 1 operation of OORTSF, that is the schedulability check report. We provide numerical output statistics and timeline display to show the result of off-line scheduling.
- Strategy: The intent of the Strategy pattern is to define a family of algorithms, to encapsulate each one, and to make them interchangeable. Strategy pattern lets the

algorithm vary independently from the clients that use the Strategy. Since different scheduling methods have different behavioral properties, we let the user explicitly choose one from the scheduling algorithms provided by OORTSF. Thus, this pattern is not needed.

In above discussions, Adapter pattern demonstrates a method for the user to integrate application tasks to the framework to make them compatible with `CTask` class. The discussions about Bridge, Iterator, Observer, Strategy, and Proxy patterns show the assessment about the design decisions used in OORTSF.

5. Extending OORTSF

OORTSF is a real-time software framework designed with embedded system as our target application domain. In OORTSF, we build the facilities of schedulability check for three well established algorithms, namely (1) cyclic executive, (2) rate monotonic scheduling, and (3) earliest deadline first scheduling algorithms. By adapting the OORTSF to an ORB of CORBA architecture, we can integrate other tasks implemented in CORBA into OORTSF. In such a way, OORTSF can be extended to behave as an ORB client application, as shown in Figure 5, and the ORB server objects can be used as application tasks of OORTSF. The timing overhead of the ORB services can be calculated and accumulated into the real-time scheduling task parameters of OORTSF `CTask` objects. After these modifications, the two phase operations of the OORTSF can be used to do the schedulability check and execution for the real-time applications on a CORBA environment. This is an example of integrating a framework to an architecture platform. We believe these kinds of interaction in high-level reuse technologies will extend the degree of software reuse.

6. Conclusion

In this article, we describe three high-level reuse techniques used in object-oriented software development: (1) design patterns, (2) frameworks, and (3) software architectures. The architecture-level reuse has been further classified into three different categories: (1) architecture styles, (2) architecture frameworks, and (3) architecture platforms. Using the design patterns to retrospect a previously designed framework will be helpful to document and evaluate the framework. With the use of an architecture platform, like CORBA, we found a way to extend the reusability of our real-time framework to a distributed computing environment. We believe that the proliferation of high-level reuse technologies: design patterns, frameworks, and software architectures, will further inspire the object-oriented software reuse to a much larger scale. The iterative examination and adaptation of high-level reuse technologies available will inspire new way of enhancing and extending the existing design in a good cost-performance ratio.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.

[2] W. Pree, *Design Patterns for Object-Oriented Software Development,* Addison-Wesley, Reading, Mass., 1994.

[3] W. Myers, "Taligent's CommonPoint: The Promise of Objects," *IEEE Computer,* Vol. 28, No. 3, March 1995, pp. 78-83.

[4] S. J. Gibbs and D. C. Tsichritzis, *Multimedia Programming: Objects, Environments and Frameworks,* Addison-Wesley, 1994.

[5] R. M. Adler, "Emerging Standards for Component Software," *IEEE Computer,* Vol. 28, No. 3, March 1995, pp. 68-77.

[6] M. Shaw and D. Garlan, *Software Architecture: perspectives on an emerging discipline*, Prentice Hall, New Jersey, 1996.

[7] D. C. Schmidt, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software," *Comm. ACM,* Vol. 38, No. 10, Oct. 1995, pp. 65-74.

[8] R. E. Johnson, "Documenting Frameworks using Patterns," In *Proceedings of OOPSLA'92,* (Vancouver, BC, Oct. 1992), pp. 63-76.

[9] F. J. Van Der Linden and Jurgen and K. Muller, "Creating Architectures with Building Blocks," *IEEE Software*, Vol. 12, No. 6, Nov. 1995, pp. 51-60.

[10] J. R. Nicol, C. T. Wilkes, and F. A. Manola, "Object Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 57-67.

[11] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20, No. 1, 1973, pp. 46-61.

[12] T. Y. Kuan, W. B. See and S. J. Chen, "An Object-Oriented Real-Time Framework and Development Environment," Position paper for the OOPSLA'95 Workshop#18, 16 Oct. 1995, Austin, Texas, USA.

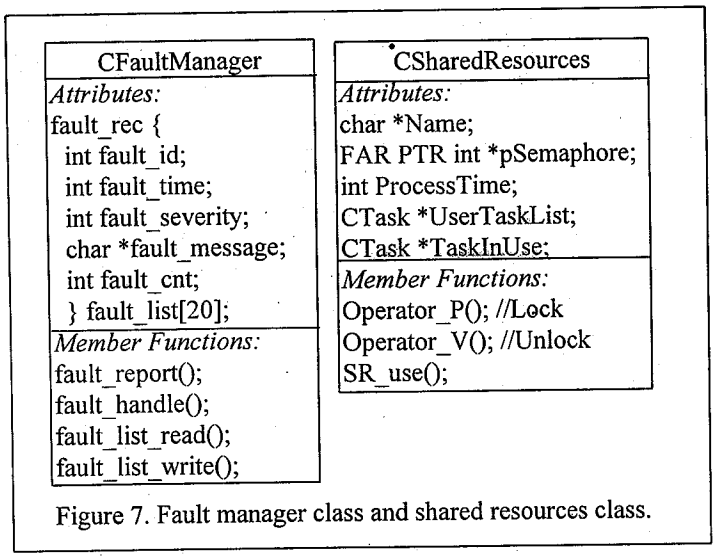

Figure 7. Fault manager class and shared resources class.

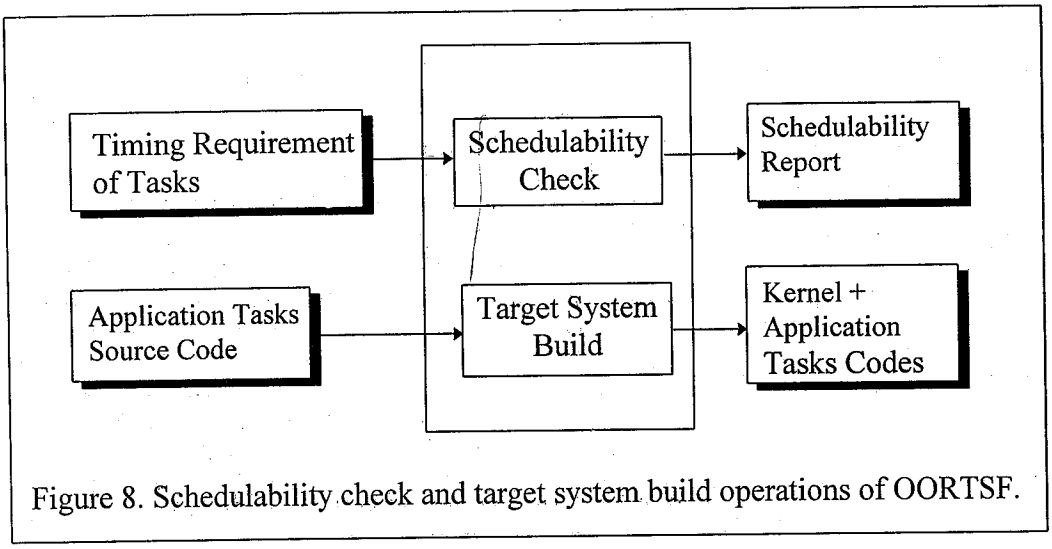

Figure 8. Schedulability check and target system build operations of OORTSF.

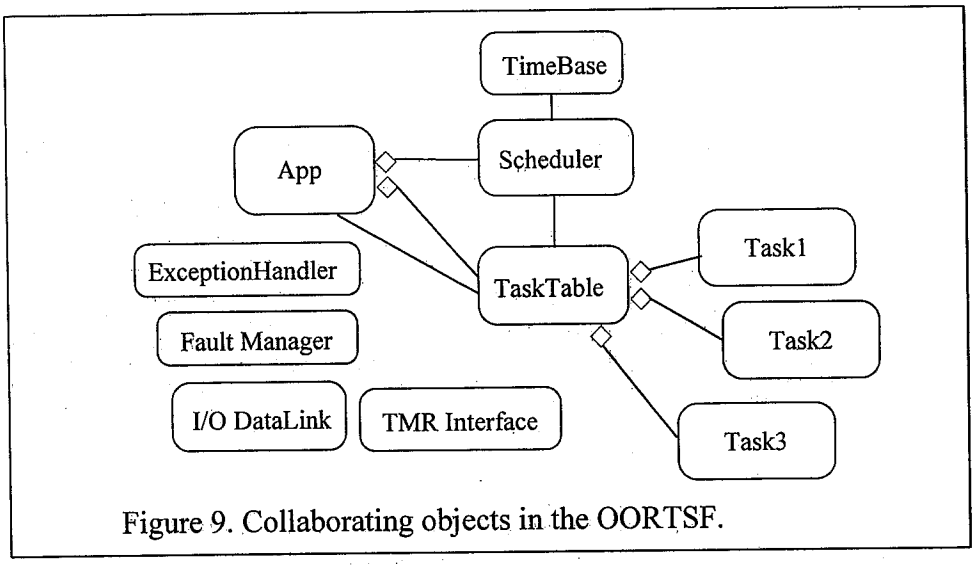

Figure 9. Collaborating objects in the OORTSF.