An Object-Oriented Approach to Software Design for Distributed Real-Time Computing Systems

Stephen S. Yau, Doo-Hwan Bae, Gil-Ho Oh*and Madhan Chidambaram†

Computer and Information Sciences Department
University of Florida
Gainesville, Florida 32611-2024, USA

Abstract

In distributed real-time computing systems for complicated applications like command, control, communications and intelligence, time constraints are severe and adaptability is required to provide high availability and survivability of computing resources. In this paper, an object-oriented approach to software design for distributed real-time systems is presented. In order to support adaptability of the software system to a dynamically changing environment, our approach supports *multi-versions* of a method definition. Our design approach is illustrated with a hypothetical air-base defense simulation system.

1 Introduction

Distributed real-time systems in applications like command, control, communications and intelligence (C3I) require complex distributed systems with many interacting software components, heterogeneous processing systems and sharing resources. These systems should satisfy not only the functional requirements of application software, but also the specified timing constraints on the execution of the software despite faults and failures. The requirements of these systems are substantially more complex than those of non-real-time systems. These systems have a high degree of complexity in terms of variety of functions, processing, storage, and communications hardware. Successful performance in real-time applications depends on satisfying the complex timing properties. Many application environments for real-time systems will be dynamically varying and somewhat unpredictable. For example, real-time systems need to be able to control, respond to, or interact with external environmental phenomenon. Such applications require the ability to adapt to changes in the environment or the external stimuli. Real-time systems must behave in a predictable manner. Current real-time systems are very expensive to build and runtime behavior is very difficult to predict at the design stage.

*Currently at Kum-Oh National Inst. of Tech., Korea
†Currently at Cap Gemini, Dayton, Ohio, USA

In this paper, we will present an object-oriented approach to software design for distributed real-time systems. The design approach is based on the object-oriented computation model for distributed real-time systems so that the real-time issues such as adaptability and timing constraints can be addressed at the proper stages of the software development life cycle. We will illustrate our design approach with a hypothetical air-base defense simulation example.

2 Background

Although the software design for distributed computing systems has been studied extensively [1]-[6], the design techniques for developing distributed real-time software systems is still in the infant stage. Current real-time software systems seem to be developed in an ad hoc fashion or based on experience in application specific domains. This complicates the program logic and structure. There are various models for real-time software systems, such as communicating finite state machines, Petri-Nets, real-time extensions of data-flow diagrams and temporal logic, but none of these models can individually represent all the aspects of a complex real-time software system. Ward and Mellor [7] developed a structured design method for real-time systems. DARTS (Design Approach for Real-Time Systems) [8, 9] is another structured design method for real-time systems. Both approaches are based on functional decomposition of software modules. In [10], object-oriented analysis and design method for real-time systems are presented. In [11], an object-oriented model is examined for its suitability to represent and encapsulate the temporal characteristics of adaptable, real-time software. However, none of these methods fully address the issues of distributed real-time software systems. We need design methodologies that can be used to synthesize systems with the specified timing properties from the design stage.

Most of the current design approaches [7, 8, 9] to real-time software systems focus on the design representation rather than the design process itself by adding constructs to specify the timing constraints. Moreover, they are not suitable for the design of distributed real-time systems for lack of design guidelines. Hence, in addition to the representation mechanisms, we need to develop design guidelines for designers to satisfy the real-time constraints by effectively utilizing the available resources. Before we discuss the design guidelines of our approach, we will first present the computation model of our approach.

3 The Computation Model of Our Approach

Our computation model is based on object-oriented concepts and incorporates the real-time system characteristics. In this model, the software system is represented by a set of objects which encapsulates the data and their related operations along with the timing constraints. We have developed the Parallel Object-Oriented Functional (PROOF) computational model [12] and a framework for the development of software for parallel processing systems [13, 14]. In this section, we will extend the concepts in PROOF to develop a model for developing distributed real-time software. We incorporate the concepts of ***active***, ***passive*** and ***pseudoactive*** objects into our model. An active object can invoke the methods of other objects. A passive object is activated by other objects when its methods are invoked. A pseudoactive object can invoke the other objects as well as be invoked by other objects. Each of the active and pseudoactive objects will have a body, which is an expression for a set of method invocations. Each object in our model will be persistent. Each object consists of local data and a set of ***methods***. Synchronization among objects is achieved by attaching an optional precondition called ***guard*** to each method. Each guard is a predicate. The object which invokes the method is suspended when the attached guard evaluates to be *False* and is resumed when the guard becomes *True*. The guard attached to a method is defined in such a way that it depends only on the local state of the object, and hence the inheritance of individual methods will not be hampered by the inclusion of the guard.

Object Modeling for Adaptability

Adaptability is an important issue in real-time software systems. To support adaptability of the software system in a dynamically changing environment, our computation model has a feature of supporting ***multiversions*** of a method definition. The communication in an object-oriented paradigm occurs through well defined interfaces via message passing. For an object to communicate with another object, the calling object needs only to know the name of the method and its parameters. In the conventional approach, each method has only one definition. In our approach, although only one method name is visible through the interface, each method can have several different versions which are not visible externally. For example, consider the following requirement where object O_a invokes another object O_p and object O_p may have to respond differently according to the state of object O_p. In the conventional approach, there are at least the following two possible design strategies:

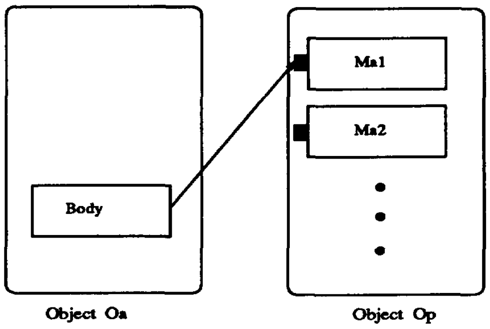

Figure 1: Multiple method definition.

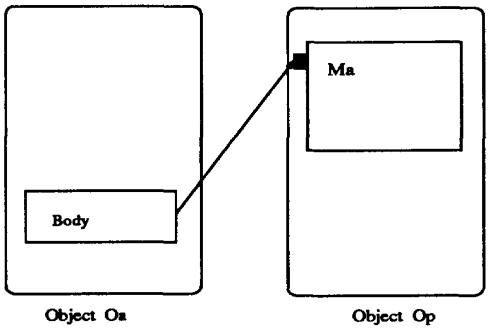

Figure 2: Single method definition with multiple actions.

- Define distinct methods in O_p for each of the different possible state of O_p as shown in Figure 1. Object O_a selects one of the methods based on the state of the object O_p. In this strategy, there is large overhead for maintaining the states of the other objects as a caller object, say O_a, needs to know the state of a called object, say O_p.
- Each of the actions to be performed on different states is encapsulated in a single method in O_p as shown in Figure 2. A particular action from this method is selected by using constructs like CASE or the IF-THEN-ELSE statements available in many high level languages. This strategy has the disadvantage that it is not modular and reduces the analyzability of the software. This strategy may limit concurrent activations of the method as the whole data could be locked by another object.

In comparison to the above two existing strategies, our approach encapsulates the actions for a state into a single method called ***virtual method***. Externally, only the name of the virtual method is visible, but internally, each action is implemented as a single method called ***actual method***. The virtual method is thus made

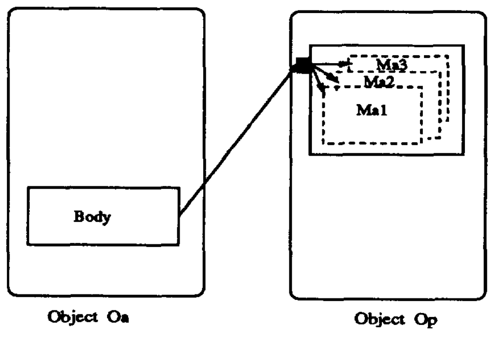

Figure 3: Multi-versions definition of a method.

up of a set of actual methods. This is illustrated in Figure 3. The state information is maintained in its local data. Depending on the current state of the object, a correct version of the method is selected for execution. Thus, the state of the object does not need to be known to the invoking objects.

Our multi-versions method approach will simplify the object interface and make the software system more readable and easily modifiable. This strategy will also enhance the parallel execution of the method since only the data that could be modified by the actual method is locked. When the state of an object is changed, for instance, from a normal state to an urgent state due to a breakdown of its neighboring object, the object may have to adapt to such a situation. For example, the object may have to change its scheduling strategy to adapt to this state change. Then, for each actual method, we can easily associate an appropriate scheduling strategy, and thus our approach can easily improve the adaptability of the real-time software at the design phase. This increases the analyzability of timing constraints at the high-level design stage, provides the designer a basis for the design of fault-tolerant software, and increases predictability of the real-time software system during the design phase since a more accurate behavior of the software system under development can be estimated.

Object Invocation Mechanism

Communication among the objects is done by message passing. To provide efficient communication for various distributed system environments, we consider both synchronous and asynchronous object invocations. To support low coupling among distributed software modules and high predictability of each software module, asynchronous remote object invocation mechanism is considered in this model. In asynchronous remote object method invocation, the invoking object will not need to wait for the remote object to be ready. One advantage of the asynchronous remote object invocation is that it will increase the concurrency among the distributed software modules. However, it may introduce buffering problems and has no big advantage in a single processor environment. On the other hand, synchronous remote object invocation will limit the concurrency. The synchronous remote object invocation can cause more tightly coupled relations among distributed software modules. In our model, synchronous object invocations will be used for the interaction among local objects and asynchronous message passing mechanisms will be used for remote object invocations.

Encapsulation of Timing Constraints in Objects

The time encapsulation mechanism is required to specify the timing constraints of the methods of each object for distributed real-time software design. To incorporate the timing constraints in our computation model, we consider the following timing attributes:

- start-time: starting execution time.
- finish-time: finishing execution time.
- duration-time: a time interval during which execution is performed.
- period: a time interval between successive executions of a periodic method invocation.

In addition to an optional guard and expression in each method, each method can have one or more optional timing constraints expression. Encapsulation of such timing constraints in each method definition will allow early evaluation of schedulability at the design phase, and can improve predictability of the software system by specifying the specific actions when the timing constraints are violated.

4 Our Approach

Our approach consists of the following steps:
1) Identify objects and classes.
2) Determine class interfaces.
3) Specify dependency and communication relationships among objects.
4) Identify active, passive and pseudo-active objects.
5) Identify the shared objects.
6) Identify periodic and aperiodic control threads.
7) Determine the priority of active and pseudoactive objects.
8) Check the completeness, consistency and schedulability of the high-level design.
9) Establish the class hierarchy.
10) Determine the body of the active objects and pseudoactive objects.
11) Design the methods of each object.

In Step 1), objects are identified by analyzing the requirements specification. Objects in the real-time systems can be classified into three types: ***input*** object, ***output*** object and ***process*** object. In a typical real-time system, the input objects provide data to the process objects for monitoring and controlling the real-time system. For instance, a temperature sensor belongs to the input object. The output objects are the objects that receive data from the process objects to physically control the system or display data to interact with the human operator. For instance, the temperature monitoring screen belongs to the output object. The process objects receive sensored data from the input objects, manipulate and send the sensored data or control signal for controlling the system.

In Steps 2) and 3), class interfaces are determined by identifying public methods, including the inputs and outputs, in each object, and then the relationships among the objects are specified by identifying the methods required by each object.

In Step 4), the objects are classified according to their invocation behavior as *active*, *passive* and *pseudoactive*.

In Step 5), the shared objects are identified from the communication relationships among the objects obtained in Step 3). Once the shared objects are determined, they can be further classified into two classes: *read-only* and *writable* objects. The distinction between the read-only objects and the writable objects is self-explanatory. Read-only objects can be duplicated as many times as desired, but writable objects cannot. Since all the access to the data in the writable objects needs to be serialized to maintain the consistency of the data, the writable objects could be a bottleneck to enhancing parallelism. Such writable objects are called *bottleneck* objects. Thus, if possible, the bottleneck objects need to be refined so as to reduce the potential of simultaneous access to the shared writable objects, resulting possible performance improvement due to the increase of parallel execution. If such refinement is done, repeat Steps 1) to 4) to make the necessary changes accordingly.

In Step 6), the periodic and aperiodic control threads are identified among the control threads. Since only the active objects can invoke the methods, all the control threads can be identified by identifying all the active objects. The periodic control thread is a thread in which the methods are invoked periodically. Most of sensory processing is periodic. For instance, a temperature monitor of a furnace in a steel manufacturing factory, and a radar to track flights. On the other hand, the aperiodic control thread is a control thread in which the methods are invoked nonperiodically. For instance, the fire-power supplier to provide more fuel to the furnace when the temperature monitor detects the temperature below a certain threshold. The characteristics of the periodic control threads, such as the periods, resource constraints, precedence relationships, communication requirements, criticalness, can be known a priori in a static system. Thus, in such a static system, the behavior of the objects involved in the periodic control threads can be accurately specified. On the other hand, in a dynamic system, such characteristics may not be statically determined at the design phase. Although it is certain that the static systems are inflexible to adjust the system behavior to unpredictable circumstances, many real-time systems are static in nature. The identification of the periodic and aperiodic control threads is important to evaluate the schedulability of the system.

In Step 7), the priority of the objects is determined according to their importance to the system. In order to determine the priority, the major functionalities of the software system need to be identified to assign the high priority. On the other hand, the objects related to minor functionalities are assigned low priority. This priority information with periodicity information obtained in Step 6) can be used to evaluate the schedulability of the software system at the design stage.

In Step 8), the completeness and consistency of the high-level design are checked against the requirements by identifying the possible scenarios of activities and examining each scenario. Because each scenario starts from the active object and the number of the active objects is not big, all the scenarios can be examined. The sequence of activities in each scenario must be reachable by tracing the behavior of the objects. If there is any object behavior which cannot be found in any of the possible scenarios, the requirements need to be re-analyzed. In addition to checking the completeness and consistency of the design, the schedulability of the real-time system under development needs to be evaluated. Although at this stage of the development, it may not be easy to obtain sufficient information to evaluate the schedulabilty. In such a case, the schedulability of periodic control threads can be checked using existing priority-based scheduling approaches. Such checking also provides a basis to roughly evaluate the schedulability of aperiodic control threads. The evaluation of the design in terms of schedulability can reduce software development effort by discovering any problems in satisfying the timing constraints at the early stage of the development.

In Step 9), the class hierarchy is established. Establishing the class hierarchy in the form of superclasses and subclasses increases the inheritance of the software. Class hierarchy also increases the modularity of the software and enhances the extensibility of the software.

In Step 10), a body is associated with each active and pseudo-active object. The role of a body is to invoke a method and modify the state of the objects represented by their local data. The modification of objects is expressed using the special construct $\mathcal{R}$ as $\mathcal{R}[|O|]e$ in which O is the object called the *recipient object* that receives a new state obtained as the result of evaluating e [12]. The modification of the objects is allowed only at the bodies of the objects. Thus, there is no side effect in the method, and history sensitivity in the object level is achieved.

In Step 11), methods in each objects are designed by selecting or creating appropriate algorithms and data structures. As we discussed in Section 3, each method can have several definitions, called multi-versions.

```
class Fighter_base

  method put_rad_value(f:Fighter_base,bomb:int,
     fght:int, miss:int, dist:int ->
                                Fighter_base)
 #called by the radar to pass the value of
 #       the enemy cluster to the base.
```

Figure 4: The class interface for *Fighter_Base*.

5 An Example

In this section, we use a hypothetical air-base defense simulation example to illustrate our approach. The specification of such a system is given as follows: *There are three air-bases: two fighter bases and one bomber base. Each base is associated with a radar, C3I(Command, Control, Communication and Intelligence) facility, air missile batteries and sufficient missiles to be used for its defense. It is assumed that each enemy cluster is composed of either missiles only or a combination of fighters and bombers, and that the enemy sends no more than two clusters to attack a base at the same time. Furthermore, the enemy cluster is assumed to target a particular base and does not change its course. Once the enemy cluster is detected by the radar, the base calculates the number of missiles or aircrafts needed to match the number of missiles or aircrafts of the enemy cluster and then sends out the required number of aircrafts and missiles. If the base cannot match the number of missiles or aircrafts of the enemy cluster with its own equipment, the base requests help from its neighboring base. When it is too late to defend the base, the aircrafts in the base will fly out of the base and go to the aircraft shelter.*

In Step 1), we identify the following objects from the requirements specification: a bomber object *b1*, two fighter objects *f2* and *f3*, three radar objects *r1*, *r2* and *r3*, a shelter object *shelter*, a record object *record* and a reporter object *reporter*. Among them, the objects *r1*, *r2* and *r3* are input objects, and the objects *b1*, *f2* and *f3* are process objects. The rest are output objects. In addition, the following classes are identified: *Bomber_class* for bomber base, *Fighter_base* for fighter base, *Radar* for radar, *Shelter* for shelter, *Record* to record the base operations, and *Reporter* to print the data store record.

In Step 2), the class interfaces are determined. The class interface for *Fighter_Base* is shown in Figure 4.

In Step 3), once the class interfaces are obtained, we establish the dependency and communication relationship among the objects. The initial object communication diagram is shown in Figure 5.

In Step 4), we identify active objects from the object communication diagram shown in Figure 5. The object *r1* does not get invoked by the other objects, but invokes *b1* and *record*. Thus, *r1* can be identified as an active object. If the communication behavior shows an object being invoked only, then the object is identified as a passive object. If the object is invoked by as well as invokes the other object, it is identified as a pseudoactive object. Thus, *r1*, *r2*, *r3* and *report* are active objects, *record* and *shelter* are passive objects, and *b1*, *f2* and *f3* are pseudoactive objects.

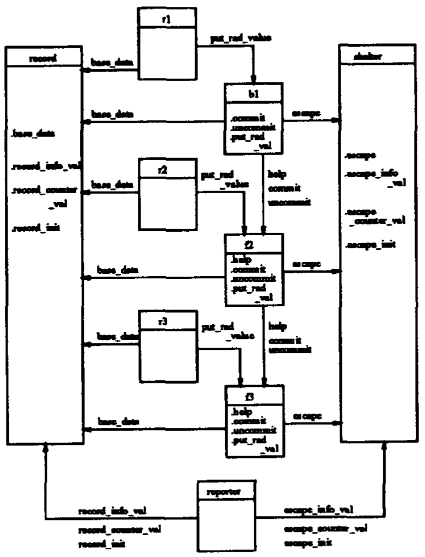

Figure 5: The initial object communication diagram for the example.

In Step 5), from the object communication diagram, we identify the objects *record* and *shelter* as shared writable objects. Since each of these objects are accessed by the three base objects *b1*, *f2* and *f3*, the access to the objects *record* and *shelter* needs to be serialized. Thus, *record* and *shelter* are bottleneck objects. We can split each of them into three objects: *record1*, *record2* and *record3* and *s1*, *s2* and *s3*. *recordi* and *si* are associated with *bi*, $i = 1, 2, 3$. The new object communication diagram reflecting such changes is shown in Figure 6.

In Step 6), from the active objects *r1*, *r2* and *r3*, periodic control threads are identified: that is, the radars generate signals periodically and give to the bases for processing. In addition, from the active object *reporter*, an aperiodic control thread is identified.

In Step 7), *r1*,*r2*,*r3*,*b1*,*f2* and *f3* are assigned high priority, and the other objects are assigned with low priority.

In Step 8), by tracing the behavior of the objects and also looking at the class interfaces, we can check the completeness and consistency of the design. In addition, we can roughly check the schedulability of the system by scheduling the tasks in periodic control threads according to the objects' priorities and then the tasks in aperiodic control threads.

In Step 9), we can define a super class called *Base*

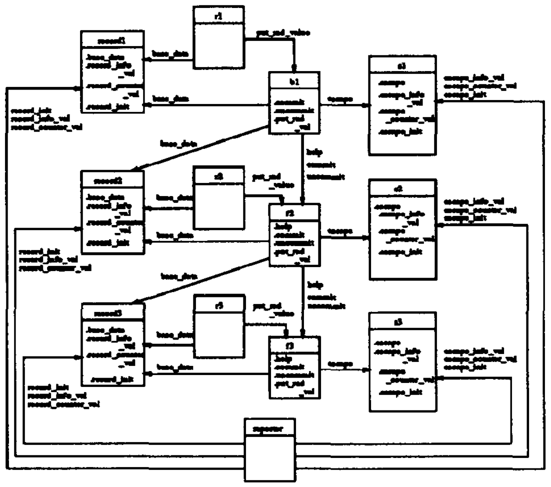

Figure 6: The object communication diagram for the modified set of objects.

which has the information common to the fighter base and the bomber base. The superclass *Base* is shown in Figure 7, *fighter_base* and *Bomber_base* are subclasses of *Base*. The remaining classes do not form a class hierarchy.

In Step 10), the body exists for only active and pseudoactive objects. Thus, the objects *r1, r2, r3, b1, f2, f3* and *reporter* have bodies. In describing the body, we use the constructs **SEQ**, **CON** and **SEL** representing sequential execution, parallel execution and selective execution, respectively. The body of *r1* is shown in Figure 8.

In Step 11), methods in each class are designed by defining a class composition, optional guards and expressions. The definition of the method **put_rad_value** in the class *Base* is shown in Figure 9. In Figure 9, we know that the method is invoked every 0.2 seconds periodically, and the method has two versions: one for the normal condition and the other for the urgent condition. In the case of the normal condition such that the enemy cluster is far away from the base, all the information about the enemy attack is given to the base. In case of the urgent condition such that the enemy cluster is so close to the base, only the distance information is given to the base.

6 Discussion

In this paper, we have presented an object-oriented approach to software design for distributed real-time systems. In order to support adaptability of the software system to a dynamically changing environment, the underlying computation model supports multi-versions of a method definition. This multi-versions method approach simplifies the object interface and makes the software system more reliable and easily modifiable. This approach can also increase predictability of the real-time software system during the design phase since a more accurate behavior of the software system under development can be obtained. To provide efficient communication, synchronous object invocation is used for interactions among the local objects, and asynchronous object invocation is used for interactions among the remote objects. In addition, the timing constraints are encapsulated in the objects so that the schedulability can be checked during the design stage. The design approach includes the steps to deal with the real-time issues, such as periodic and aperiodic tasks, priorities of the objects, and schedulability of the high-level design.

We will continue to develop a high-level design description language based on our computation model to support systematic design of distributed real-time systems. It will have both textual description capability and also a graphical representation capability. This will be appropriate for analysis and verification of distributed real-time systems. We will also develop a strategy to transform the design description language into a timing analysis model such as the timed petri-nets to facilitate the analysis of timing constraints.

```
class Base

  method put_rad_value(f:Base, bomb:int,
    fght:int, miss:int, dist:int -> Base)
#called by the radar to pass the value of
#     the enemy cluster to the base.

  method compute_range ( f:Base -> int )
#returns a value proportional to the range.
#such functions are used to for the sake of
# functional programming style.

  method enemy_cluster (f:Base -> int)
#determines whether a missile or aircraft attack.

  method effective (f:Base -> int)
#effectiveness of the enemy cluster


#to keep track of aircrafts currently on ground.

  method commit ( f:Base, n:int -> Base)
  method uncommit ( f:Base, n:int -> Base)

end class
```

Figure 7: The superclass *Base*.

References

[1] S. S. Yau, C. C. Yang, and S. Shatz, " An Approach to Distributed Computing System Software Design," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 4, July 1981, pp. 427-436.

```
SEQ( #Assign values to the radar object
 R[| r1 |] object radar( pre_assign radar
                                   values),
  while(True,
  #Generate the radar values to be passed
  #on to the base.
    CON(r1.generate_rad_bombervalue,
        r1.generate_rad_fightervalue,
        r1.generate_rad_missilevalue,
        r1.generate_rad_distancevalue),
  #Put the values obtained above in the base
  #and record, and then
  #modify the radar values.  This operation
  #will modify all the objects involved.
    CON( R[| b1 |] b1.put_rad_value,
         R[| record1 |] record1.base_data,
         R[| r1 |] r1.modify_list)
        )
    )
```

Figure 8: The body of *r1*.

```
class Base
  compostion
   radar_bomber:int
   radar_fighter:int
   radar_missile:int
   radar_distance:int
      ...
      ...

  method put_rad_value(b:Base, bomb:int,
    fght:int, miss:int, dist:int -> Base)
    period: 0.2s
    guard:none
    version 1: dist > 50
      expression
        radar_bomber = bomb
        radar_fighter = fght
        radar_missile = miss
        radar_distance = dist
    version 2: dist <= 50
      expression
        radar_distance = dist
          ...
          ...
end class
```

Figure 9: The *put_rad_value* method definition.

[2] S. S. Yau, M. U. Caglayan, " Distributed Software System Design Representation Using Modified Petri Nets," *IEEE Trans. on Software Engineering,* Vol. SE-9, No. 6, Nov. 1983, pp.733-745.

[3] C.K. Chang, et al., " A New Design Approach of Real-Time Distributed Software Systems," *Proc.. 11th International Computer Software & Applications Conference,* (COMPSAC 87), October 1987, pp. 474-479.

[4] S. S. Yau, and I. Wiharja, " An Approach to Module Distribution for the Design of Embedded Distributed Software Systems," *Journal of Information Sciences,* Vol. 56, August 1991, pp. 1-22.

[5] S. S. Yau, X. Jia, and D.-H. Bae, " On Software Design Methods for Distributed Computing Systems," *Computer Communications Journal,* Vol. 14, May 1992, pp. 213-223.

[6] H. Kopetz, et al, "Real-Time System Development: The Programming Model of MARS," *Proc. Int'l Symp. on Autonomous Decentralized Systems,* April 1993, pp. 290-299.

[7] P.T. Ward, and S. J. Mellor, " Structured Development of Real-Time Systems," (Volumes 1, 2, 3) Yourdon Press, 1985.

[8] H. Gommaa, " Software Development of Real-Time Systems," *Comm. ACM,* Vol. 29, No. 7, July 1986, pp. 657-668.

[9] H. Gommaa, " A Software Design Method for Real-Time Systems," *Comm. ACM,* Vol. 27, No. 9, September 1984, pp. 938-949.

[10] M. Baldassari, and G. Bruno, " A Methodology and Environment for the Object Oriented Analysis and Design of Real Time Systems," *Proc. Euromicro'90 workshop on Real Time,* June 1990, pp. 72-78.

[11] T. Bihari, P. Gopinath and K. Schwan "Object-oriented Design of Real-Time Software," *Proc. Real-Time Systems Symposium,* December 1989, pp. 194-201

[12] S.S. Yau, X. Jia and D.-H. Bae, "PROOF: A Parallel Object-Oriented Functional Model," *Journal of Parallel and Distributed Computing,* Vol. 12, No. 3, July 1991, pp. 202-212.

[13] S.S. Yau, X. Jia, D.-H. Bae, M. Chidambaram, and G. Oh, "An Object-Oriented Approach to Software Development for Parallel Processing Systems," *Proc. 15th International Computer Software & Applications Conference,* (COMPSAC 91), September 1991, pp. 453-458.

[14] S.S. Yau, D.-H. Bae and M. Chidambaram, "A Framework for Software Development for Distributed Parallel Computing Systems," *Proc. Workshop on Future Trends of Distributed Computing Systems in the 1990's,* April 1992, pp. 240-246.