Copyright 1994 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Proc. WORDS '94 (IEEE Computer Society Workshop on Object-oriented Real-time Dependable Systems), Dana Point, Oct. ‘94, pp.36-45.

Distinguishing Features and Potential Roles of the RTO.k Object Model

K. H. (Kane) Kim, Luiz Bacellar, Yuseok Kim,
University of California, Irvine

Dong K. Choi,
University of Pennsylvania

Steve Howell, and Michael Jenkins
USN Naval Surface Warfare Center

Abstract: In recent years, search for proper extension of the basic object model to meet the needs present in the hard-real-time system development environments has become a serious research issue. The first co-author and Hermann Kopetz at Technical University of Vienna, formulated an extension of the basic object model as one attempt to meet such needs and it has been called the RTO.k object model. In the past two years, we have been making efforts to develop practical easy-to-use tools which assist the system engineers in
(1) RTO.k structured description and simulation of application environments and
(2) RTO.k structured hierarchical design of control computer systems.
In this paper, unique features of the RTO.k model which distinguish it from other extensions of the basic object model as well as common features will be presented first. The roles which the RTO.k model can play during various steps of the real-time system engineering process will then be discussed.

1. Introduction

Almost from the beginning days of object-oriented structuring [Dah72, Boo91, Rum91], numerous engineers and researchers concerned with real-time computing applications thought about introducing deadlines into the basic object model. However, system engineers have not been able to produce convincing demonstrations of the significant improvements in development of hard-real-time distributed computer systems (DCS’s) with such a simply extended structuring approach. We feel that the most significant and desirable advance in the art of hard-real-time DCS development will have occurred when design-time guarantee of timely service capabilities of a system becomes an integral and easily practiced step in system engineering. Therefore, search for proper extensions of the basic object model to meet such and other needs present in the hard-real-time system development environments has become a serious research issue.

A seemingly distinct but closely related motivation to find a proper extension of the basic object model arises from the need to establish a coherently integrated methodology for engineering real-time DCS’s [Kim93]. We perceive that the following conditions exist in the current practice of real-time system engineering:
(1) Lack of rigor in requirements specification, in particular, specification of temporal behavior requirements and dependability requirements;
(2) Weak traceability among various specifications and system models used during high-level design, implementation, validation, and evaluation; and
(3) Lack of integration in design techniques with poor consequences in the dependability and the guaranteed response delay of the systems produced.
In our view, one of the missing cornerstones for developing a coherently integrated engineering methodology is a system (and component) model which is effective not only for abstraction and stepwise refinement of real-time computer system designs but also for representing and providing a basis for analysis of the application environments. Not only descriptions but also simulations of application environments are often performed as integral steps of validating control computer system designs. A desirable model is thus one that supports realization of the *uniformity* and the high degree of *accuracy* in representation of application environments, environment simulators, and control computer system designs at different levels evolving during the system development cycle.

The RTO.k object model, also called the *time-triggered real-time object* (TT-RTO) model, is a result of the attempt by the first co-author and Hermann Kopetz at Technical University of Vienna to find a proper extension of the basic object model which is capable of uniformly and accurately representing both embedded computer systems and application environments. Based on the initial framework formulated by Kopetz and Kim in late 1980’s [Kop90], a concrete syntactic structure associated with unambiguous execution semantics has been produced in recent years [Kim93, Kim94b]. The ease of facilitating design-time guarantee of timely service capabilities of objects has also been used as the fundamental guiding principle in devising this first concrete version of the RTO.k object model. Two most distinguishing characteristics of this model relative to other proposed extensions of the basic object model are (1) the clear-cut separation of the *time-triggered object methods*, also called the *spontaneous methods*, from the conventional,

message-triggered object methods, also called the *service methods*, and (2) the execution rule called the *basic concurrency constraint*.

Under the RTO.k object based uniform structuring approach, the combination of a control system design and an environment simulator takes the form of a network of RTO.k objects. A specification and implementation experiment that involved an application of the RTO.k structuring scheme to both the development of a defense system and that of an environment simulator was conducted recently [Kim94b]. This experiment reinforced our belief that the RTO.k model had the necessary representational power and also offered an efficient and rigorous way to develop complex real-time systems.

An overview of the major features of the RTO.k model is given in the next section and then the distinguishing features of the RTO.k model not found in other proposed extensions of the basic object model are discussed in Section 3 together with the costs and benefits of those features. Potential useful roles of the RTO.k object model in engineering of complex real-time DCS's are discussed in Section 4. The paper concludes in Section 5.

2. Major features of the RTO.k model

The following two goals have been adopted as the fundamental guiding principles in formulation of the current RTO.k object model.

(a) Uniform structuring of both real-time DCS's and their application environment simulators;
(b) Facilitating design-time guarantee of timely service capabilities of objects.

2.1 Major extensions of the basic object model

As an extension of the conventional object model(s) the RTO.k object model is :
a) independent of the language (textual or graphic) used to program or specify object designs, and
b) independent of the way inheritance is facilitated.
Since one can envision the emergence of multiple concrete language constructs based on the current RTO.k object model, the model discussed here can be viewed as a framework for an evolving model family.

The basic structure of the RTO.k object model is depicted in Figure 1. The RTO.k object model is an extension of the conventional object model(s) in four essential ways:
(a) For some methods of an RTO.k object, a real-time clock serves as the mechanism for triggering the method executions as the clock reaches some values specified at the design time and such methods are called time-triggered (TT-) methods, also called the spontaneous methods (SpM's), and *clearly* separated from the conventional service methods (SvM's) triggered by messages from clients. Actions to be taken when the real-time clock reaches some values *which can be determined at the design time* can appear only in SpM's;
(b) A concurrency constraint which prevents conflicts between TT-methods and message-triggered methods is incorporated. Basically, *activation of a service method triggered by a message from an external client is allowed only when potentially conflicting TT-method executions are not in place.* To be exact, when a message-triggered service method is not free of conflict in accessing the same portion of the object data space (ODS) with a TT-method, execution of the former (message-triggered) method must not be allowed in a time zone earmarked for a TT-execution of the latter (spontaneous) method. This restriction is called the basic concurrency constraint (BCC). Therefore, TT-methods are given higher priorities for execution over the message-triggered methods. Note that this BCC does not impose any restriction on concurrent execution of TT-methods or concurrent execution of message-triggered methods;
(c) For each execution of a method of an RTO.k object, a deadline is imposed;
(d) Real-time data contained in an RTO.k object become invalid after the interval called the maximum validity duration passes.

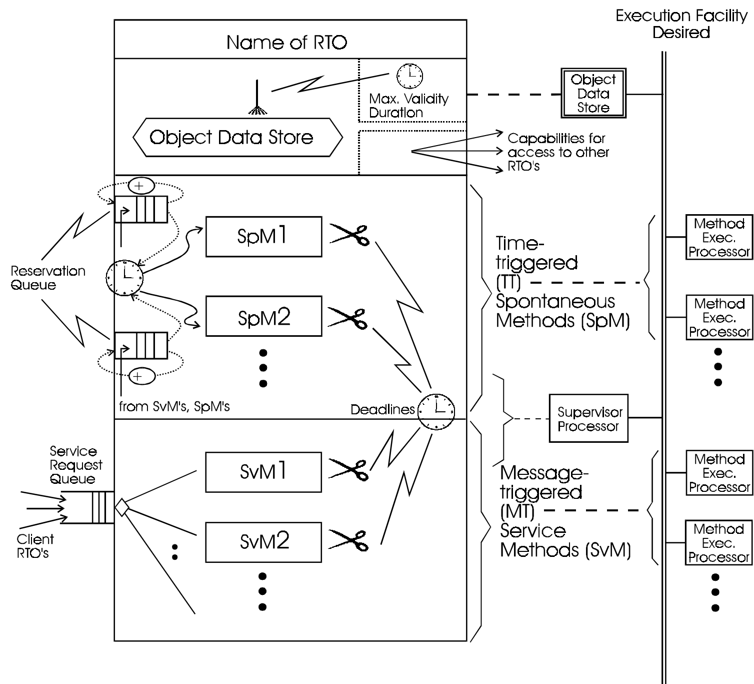

Figure 1. Structure of the RTO.k object model

In Figure 1, the two distinct types of methods for operating on its *object data space* (also called the *object data store*), the SpM's and the message-triggered SvM's, are shown. The clear-cut separation of SpM's from SvM's is an important feature of the RTO.k object model.

The two types of methods are different not only in the way their executions are triggered but also in that actions of the type "at constant-clock-value do S" or the type "sleep-until constant-clock-value" can appear only in SpM's. For example, if a value computed by an SvM needs to be output precisely at a certain time, then the RTO.k object must be designed such that a request is made by the producer SvM to an SpM to carry out the output action. The exact mechanism by which such a request is conveyed will become clearer in the next subsection.

2.2 Autonomous activation condition (AAC)

Triggering times for SpM's must be *fully specified as constants* during the design time. Those real-time constants appear in the first clause of an SpM specification called the autonomous activation condition (AAC) section. The AAC may be specified in the following form.

```
ab      "AAC-begin"
{ [AAC name:]
    for <time-var> = from <activation-time>
            to <deactivation-time>
            [every <period>]
    start-during
        (<earliest-start-time>,
          <latest-start-time>)
    finish-by<deadline>
}*
ae      "AAC-end"
```

where the "star" expression x* or {x}* is a regular expression for the set {NULL, x, xx, xxx, ...}.

For example, consider the following case.

```
ab
AAC-1:
    for T =  from 10:00:00.000am
        to 11:00:00.000am
        every 0.005sec
    start-during  (T, T+0.001sec)
    finish-by      T+0.003sec
ae
```

The above AAC-1 specifies : "This SpM must be executed every 5 msec starting at 10 am until 11 am and each execution must start at any time within the one millisecond interval (T, T + 0.001sec) and must be completed by T + 0.003sec."

Note that "for t = from 10am to 10am start-during (t, t+5min)" has the same effect as "start-during (10am, 10:05am)". Note also that the AAC section may contain multiple unnamed or uniquely named AAC clauses.

The execution engine uses the AAC of an SpM to generate future execution triggering schedules for the SpM at appropriate times and inserts them into the reservation queue (depicted in Figure 1) of the SpM. Each reservation queue is a sorted queue in which the nearest future triggering schedule is at the head of the queue.

A provision is also made for making the AAC section of an SpM contain only candidate triggering times, not actual triggering times, so that a subset of the candidate triggering times indicated in the AAC section may be dynamically chosen for actual triggering. Such a dynamic selection occurs when an SvM within the same RTO.k object requests future executions of a specific SpM. Again, candidate triggering times must be fully specified as constants during the design time. An SvM requests future executions of an SpM by placing a reservation into the reservation queue associated with the SpM. The reservation queue holds future execution triggering schedules for the SpM determined up to the present. To simplify the job of the compiler, a reservation by an SvM for execution of an SpM must include the name of an AAC clause appeared in the SpM definition. The part of the AAC section containing AAC clauses that specify candidate triggering times rather than actual triggering times starts with a declaration "if-demanded". For example,

```
ab    if-demanded
a1:  start-during  (10am, 10:05am)
        finish-by  10:15am;
a2:  start-during  (11am, 11:05am)
        finish-by  11:15am   ae
```

defines two candidate triggering times. An SvM may then call this SpM, say SpM-1, as follows.

```
if current-time < 9:30am
    then  request SpM-1(AAC=a1)
    else  request SpM-1(AAC=a2).
```

Therefore, there are two different modes of determining triggering times for SpM's:
(a) fully determined during the system design, in which case the SpM is said to be statically scheduled, and
(b) determined during the run time when an SvM requests executions of the SpM and designates a subset of the candidate triggering times prepared during the design time as actual triggering times, in which case the SpM is said to be partially dynamically scheduled.

On the other hand, actions to be taken when the real-time clock reaches values which cannot be determined at the design time may appear in SvM's, even if the actions may have to be executed periodically. Therefore, an SvM may include a statement of the type "for X seconds every Y seconds do S". However, the start time of this statement execution is not known until the SvM is called by a client.

2.3 Interactions among RTO.k objects

An underlying design philosophy of the RTO.k object model is that a real-time DCS will always take the form of a network of RTO.k objects. RTO.k objects interact via calls by client objects for SvM's in server

objects. The caller may be an SpM or an SvM in the client object. In order to facilitate highly concurrent operations of client and server objects, non-blocking (sometimes called asynchronous) types of calls (i.e., service requests) in addition to the conventional blocking type of calls can be made to SvM's.

Therefore, the following two basic types of calls can be made to SvM's in the server RTO.k object.

(a) Blocking call: After calling an SvM, the client waits until a result message is returned from the SvM. The syntactic structure may be in the form of

Obj-name.SvM-name(parameter-1, parameter-2, ...).

Since the client and the server object may be resident in two different processing nodes, this call is in general implemented in the form of a remote procedure call. Even if there is no result parameter in the SvM, the execution completion signal is returned to the client;

(2) Non-blocking call: After calling an SvM, the client can proceed to follow-on steps (i.e., statements or instructions) and then waits for a result message from the SvM. The syntactic structure may be in the form of

Obj-name.SvM-name(parameter-1, parameter-2, .., mode NWFR, timestamp TS);
----- statements -----
get-result Obj-name.SvM-name(TS) by deadline;

The mode specification "NWFR" which is an abbreviation of "No-Wait-For-Return" indicates that this is a non-blocking call. When the client calls the SvM, the client records a time-stamp into a variable, say TS. The time-stamp uniquely identifies this particular call for the SvM as distinct from other (past or future) calls for the same SvM from this client. Therefore, later when the client needs to ensure by execution of the "get-result" statement the arrival of the results returned from the earlier non-blocking call for the SvM, not only the SvM-name but also the variable TS containing the time-stamp associated with the subject call must be indicated. When a client makes multiple non-blocking calls for SvM's before executing a "get-result" statement, the time-stamp unambiguously indicates to the execution engine which non-blocking call is referred to.

If the results have not been returned at the time of executing the "get-result" statement, the client waits until the execution engine recognizes the arrival of the results. A non-blocking call thus creates concurrency between a client (SpM or SvM) and a server SvM which lasts until the execution of the corresponding "get-result" statement. In some situations, a client does not need any result from a non-blocking call for an SvM. Such a client does not use a "get-result" statement.

2.4 Concurrency and working ODS specification

The following types of concurrency can be exploited in execution of object methods in an RTO.k object:

(a) Concurrency among SpM executions;
(b) Concurrency among SvM executions;
(c) Concurrency between SpM executions and SvM executions.

Concurrency among the SpM's is specified in an implicit but natural manner, e.g., two SpM's designed to be triggered at 10 am. In general, if the specified execution periods of two SpM's are in overlap, then there is clearly concurrency implied in the two SpM specifications.

The approach adopted in the RTO.k object model for exploiting concurrency of type-b and type-c is to explicitly declare the portion of the object data space (ODS) used by each method and allow concurrent execution of methods whenever there is no data conflict. Therefore, the ODS-section in the RTO.k class consists of declarations of ODS-segments (ODSS's). Each ODS-segment is explicitly named. The specification of each object method then includes its working ODS specification which indicates the set of ODS-segments that can be operated on during the execution of the method. This specification also indicates if each ODS-segment can be accessed for "read-only" or "read-&-write" purpose. This facilitates detection of potential data conflicts among the object methods that need to be executed. Each ODS-segment is thus an atomic storage unit specified explicitly as such by the designer. The sizes of such ODS-segments determine the degree of parallelism that can be exploited in execution of object methods.

During the design time the working ODS specifications are used for detection of errors in SpM specifications. If two SpM's are specified for concurrent execution while both SpM's have "read-&-write" privileges for the same ODS-segment(s), then the SpM specifications are incorrect.

During the execution time the working ODS specifications are used by the execution engine to determine whether SvM's requested by clients can be initiated immediately or should be delayed until some potentially conflicting methods are completed. For example, suppose the working ODS specification of an SvM, SvMx, consists of {(ODSS-1, RW)} where RW and RO indicate the "read-&-write" right and the "read-only" right, respectively. Once a client calls for SvMx and the execution engine starts evaluating the possibility of initiating the SvMx execution, the engine checks if any SpM or SvM currently in execution has an access right (RO or RW) for ODSS-1. If so, then SvMx cannot be initiated yet. Otherwise, the engine checks if any SpM which will have an access right for ODSS-1 is going to be triggered before t + MET(SvMx) where t represents the current time and MET(SvMx) represents the maximum execution time estimate for SvMx. Again, if so, SvMx cannot be initiated yet. Otherwise, SvMx is initiated.

To facilitate exploitation of additional concurrency among SvM executions at the cost of increased burdens on the object designer for determining the worst-case service time, access modes for the working ODS-segments of SvM's may be "unspecified" and instead, each ODS-segment may be encapsulated within a CREW (concurrent-read-&-exclusive-write) monitor which is an

extension of the monitor in [Bri73] and possesses the readers-writers semantics [Bri73]. In this case, an SvM called by a client can be initiated as long as there is no potential data conflict with an SpM. Potential data conflicts with another SvM can be ignored since shared ODS-segments are safely protected within CREW-monitors.

Suppose three object methods in the same RTO.k object have the following working ODS specifications.

Working ODS of SpM1: {(ODSS1, RO),
(ODSS2, RW)}
Working ODS of SvM1: {(ODSS1, unspecified)},
Working ODS of SvM2: {(ODSS1, unspecified)}.

Suppose also that SvM1 has been called by a client, the current time is 10am, SpM1 is to be triggered at 10:05am, and MET(SvM1) is 10 minutes. SvM1 cannot be initiated at this time because the access mode of SvM1 for ODSS1 is unspecified and thus can be RW and also because if SvM1 is initiated now, there is possibility of the SvM1 execution being unfinished at 10:05am at which time SpM1 will be triggered. Suppose SpM1 is completed at 10:10am. At this time, SvM1 is initiated. If SvM2 is called by a client at 10:12am and SvM1 is still in execution at this time, SvM2 can be initiated as long as there is no potential data conflict between SvM2 and an SpM. This is because ODSS1 is encapsulated within a CREW-monitor and thus can only be shared harmoniously by SvM1 and SvM2.

2.5 Design-time guarantee of timely service capabilities of RTO.k objects and specification of maximum service times

An important underlying design philosophy of the RTO.k object model is that a real-time DCS will always take the form of a network of RTO.k objects and

each RTO.k object should be maximally autonomous in timing its actions while providing dependable services to client RTO.k objects.

Therefore, the designer of an RTO.k object can view SpM's as internal capabilities and SvM's as services advertised to all potential clients.

The designer of each RTO.k object provides a guarantee of timely service capabilities of the object by indicating the *deadline for every output* produced by each SvM (and each SpM which may be executed on requests from SvM's) in the specification of the SvM (and some relevant SpM's) advertised to the designers of potential client objects. An output action here may be

(a) an updating of a portion of the ODS,
(b) sending a message to either another RTO.k object (which may or may not be the client) or a device shared by multiple objects, or
(c) placing a reservation into the reservation queue for a certain SpM.

The specification of each SvM which is provided to the designers of potential client RTO.k objects must contain at least the following:

(a) an *input specification* that consists of
(a1) the types of input parameters that the server object can accept and
(a2) the maximum request acceptance rate, i.e., the maximum rate at which the server object can receive service requests from client objects;
(b) an *output specification* that indicates the *maximum delay* (not the exact output time) and the *nature of the output value* for every output produced by the SvM. In a sense, the maximum among all the maximum delays associated with output actions expected from the SvM is the maximum service time of the SvM.

If service requests from client objects arrive at a server object at a rate exceeding the maximum acceptance rate indicated in the input specification for the server object, then the server may return exception signals to the client objects. The system designer can prevent such "overflow" occurrences by careful design or provide exception handlers if he/she is certain that exception handlers will never fail to achieve the application goals satisfactorily (in absence of component failures).

The specification of the maximum delay for an output from an SvM is a serious commitment on the part of the server object designer to the designers of potential client objects. It is a *guarantee of timely service*. Before determining the maximum delay specification, the server object designer must convince himself/herself that with the object execution engine (hardware plus operating system) available, the server object can be implemented to always execute the SvM such that the output action is performed within the maximum delay (in absence of component failures). This means that the server object designer must consider

(1) the worst-case delay from the issuance by a client object of a service request to the initiation of the corresponding SvM by the server object, and
(2) the worst-case execution time for the SvM from its initiation to each of its output actions.

On the other hand, a client RTO.k object imposes a deadline on the server RTO.k object for creation of all the intended computational effects (i.e., all intended output actions). The deadline imposed by a client must not be earlier than the deadline adopted and advertised by the designer of the server object for completion of the corresponding SvM.

The specifications of the SpM's which may be executed on requests from the SvM must also be provided to the designers of the client objects which may call the SvM. The specification of such an SpM must contain at least the following:

(1) an *autonomous activation condition (AAC)*, and
(2) an *output specification*.

There is no input specification in an SpM specification but the output specification for an SpM indicates, for every output expected from the execution of an SpM, the exact time *at* or *by* which it will be produced and the nature of every value carried in the output action. As an example, one SpM can be designed to "update a specific data item

in the object data store" (nature of output) "at 10:30am (time for output) if the SpM started its execution between 10:10am and 10:15am.

The basic concurrency constraint (BCC) was incorporated into the RTO.k object model to ease the design-time guarantee of timely service capabilities. At least it makes it very easy to analyze the execution time behavior of SpM's. Executions of SpM's are not disturbed by SvM executions and triggering times of SpM's are fixed at the design time. For example, if a statement of the type "at 10am do S" appears in an SpM, its reliable execution can be easily assured. Therefore, if most of the computations are done by SpM's and only simple client interface functions are handled by SvM's, then it should be easier to guarantee timely service capabilities than in the cases of objects in which most computations are done by SvM's and frequent competition among SvM's occurs for access to the same portion of the ODS.

2.6 Maximum validity duration

The ODS-section consists of declarations of ODS-segments and each ODS-segment declaration consists of variable declarations. Here maximum validity durations (MVD's) can be associated with individual variables as follows.

<type-specifier> <identifier> [during <effective-period> | by <expiration-time>]

For example,

"int K during 5 msec"

indicates that a new value stored into the integer-type variable K will be valid for use only for 5 milliseconds. The MVD specification can be used by error detectors planted in the execution engine.

2.7 Overall structure of the object method specification

A model of a language construct for structuring RTO.k objects, the *RTO.k class*, was defined in [Kim94b]. An RTO.k class definition consists of four major sections.

RTO_class = class
begin
    "connectivity_section :"
        list of RTO_name;
            "that can be called upon"
    "object_data_space_section :"
        list of object_data_space_segment;
    "spontaneous_method_section :"
        list of spontaneous_method;
    "service_method_section :"
        list of service_method
end;

The overall structure of the SpM specification in an RTO.k class may be as follows.

SpM-name *<name>*
    using-SpM *<SpM-name>**
    using-data (*<ODS-segment-name>*, *<access-mode>*)*
    < AAC section >
    finish-by *<deadline>*
begin
    <method-body>
end;

Here the "using-SpM" section lists the names of SpM's within the same RTO.k object to which activation requests can be sent by this SpM.

The overall structure of the SvM specification in an RTO.k class may be as follows.

SvM-name *<name>* (*<parameter specification>*)
    using-services *<RTO-name.SvM-name>**
    using-SpM *<SpM-name>**
    using-data (*<ODS-segment-name>*, *<access-mode>*)*
    [finish-within *<duration-limit>* |
     finish-by *<deadline>*]
begin
    <method-body>
end;

Here timely completion requirements can be specified in two different ways but the specification of the method completion deadline is an expression which cannot be fully evaluated until the execution-start time of the SvM is determined.

2.8 Extended mode interactions among RTO.k objects and among object methods within the same object

In addition to the two basic types of interactions among RTO.k objects, the blocking-call for an SvM and the non-blocking call for an SvM, a major variation of each of the two types is also adopted into the RTO.k object model for the sake of reduced communiction overhead in some situations. The essence of this extension is to allow an arrangement in which a client calls an SvM and then receives results from another SvM. The main motivation for facilitating this stems from the basic concurrency constraint which requires an execution of any SvM to be made on a stand-by basis, i.e., only when a sufficiently large time window between SpM executions opens up.

Therefore, if the maximum execution time estimate of an SvM, say SvMx, is MET(x) seconds and the SpM's are so frequently executed that a time window larger than MET(x) never opens up, then the SvM may never be executed. One way to get around this problem is to divide such an SvM, SvMx, into multiple smaller SvM's, SvMx1, SvMx2, ---, SvMxk. A client must then call each

smaller SvM at a time. Calling each smaller SvM incurs communication overhead for transmitting a call message from the client to the SvM and a result message from the SvM to the client. In general, these inter-object messages can involve much larger delays than intra-object messages. Naturally, one can conceive of an arrangement in which a client calls the first SvM (SvMx1), the latter in turn "passes its client" to the second SvM (SvMx2) as it completes its execution, and this continues until the last SvM (SvMxk) is executed and then returns the results to the inherited client which is the external client that called the first SvM. Passing the client from an SvM to the next SvM involves intra-object messages although such a client-transfer call message goes through the same service request queue which service requests from external clients go through.

A client-transfer call may involve passing parameters in an explicit manner as done in the case of a call by an external client or passing information through the shared data structures in the ODS. The syntactic structure for such a client-transfer call for an SvM may be in the form of

SvM-name(parameter1, parameter2, ---, mode CT,
external parameter-x1, parameter-x2, ---).

The mode specification "CT" which is an abbreviation of "Client Transfer" indicates that this is a client-transfer call. The parameters listed in the "external" clause are those which were used in the interface between the caller and the caller of the caller. Normally those parameters are inherited from the interface between the external client (i.e., located outside the subject server object) and the first SvM called by the client.

As a part of executing this client-transfer call for an SvM, the execution engine terminates the caller SvM, places a request for execution of the called SvM into the service request queue, and establishes the return connection (i.e., the connection to the client) from the called SvM to the client of the caller SvM that has just been terminated. When the "return" statement in the called SvM is executed, the results are returned through the return connection established. Since the external client which called the first SvM cannot predict from which SvM it will receive returned results, it must be implemented to accept results returned from any SvM.

There is no reason why this client-transfer call cannot be extended to the case of calling an SvM in another RTO.k object. The syntactic structure for such a client-transfer call for an external SvM may be in the form of

Obj-name.SvM-name(parameter1, parameter2, ---,
mode CT, external parameter-x1, ---).

Besides the above client-transfer call for an external SvM, no other language constructs need be introduced due to the adoption of the extended-mode interaction among RTO.k objects. This is because accepting results returned from the called SvM is a special case of accepting results returned from any SvM in the system. Moreover, under our philosophy of making the details of a server object transparent to the designer of a client, the client calling an external SvM need not be able to distinguish between the case where results are returned from the called SvM and the case where results are returned from another SvM.

3. Distinguishing characteristics of the RTO.k object model

In this section, major differences between the RTO.k model and other proposed extensions of the basic object model are discussed together with the costs and benefits which each distinct feature brings to the RTO.k model.

3.1 Clear-cut distinction between SpM and SvM

All real-time extensions of the basic object model proposed so far are active objects with some time constraints added. An active object is an object with its own thread of execution control. Therefore, multiple active objects can exhibit concurrency among their activities. Of all the models proposed as real-time extensions of the basic object model, about half of them do not provide anything resembling SpM's. Such models will not be discussed any further in this paper [Bih89, Cha90, Wol91]. In the object model adopted in the RTC++ project [Ish90, Ish92] and other models [Att91, Her92, Shi91], the clear-cut distinction made in the RTO.k model between SpM's and SvM's is not done. That is, the rule, "actions to be taken when the real-time clock reaches values which can be determined at the design time can appear only in SpM's", is not adopted in any of those models. This rule was adopted in the RTO.k to simplify the task of guaranteeing the timely service offered by the RTO.k object. Also, the number of SpM executions that can proceed in parallel has no fixed limit in the RTO.k model unlike in models such as the MO2 model [Att91] and others [Shi91].

In some models providing something similar to SpM's, interactions between those corresponding to SpM's and SvM's are not facilitated [Her92, Shi91, Tak92].

3.2 Basic concurrency constraint

The MO2 model proposed in [Att91] is the only other model which contains something resembling the basic concurrency constraint. However, the MO2 model allows only one SpM in an object. Also, an SvM in an RTO.k object is not initiated if it has the potential of running into data conflicts with any SpM in execution or with any SpM scheduled. On the other hand, SvM's and the SpM in an MO2 object may be in concurrent execution with intermittent competition for accessing data in the ODS. Therefore, the RTO.k model sacrifices some fine degree of parallelism for the sake of ease in guaranteeing timely services of objects.

3.3 Design-time guarantee of timely service of each object

The RTO.k object model was formulated with this specific objective of facilitating design-time guarantee of timely services of each object in mind. For example, the AAC section in the SpM declaration is restricted to be an expression which can be fully evaluated at design time. The execution engine which contains the operating system and both the inter-object communication facility and the intra-object inter-method communication facility, is required to yield easy analysis of the worst-case execution time for any local or remote method execution. It appears that this "conservative" policy was not adopted in any other model, or at least not pursued to the extent done in the RTO.k model. However, the conservative policy of the RTO.k model can result in some sacrifice of hardware utilization in comparison to the case of using "liberal" policies adopted in other models.

Having recognized the desirable characteristics of the execution engine, an execution engine model called the DREAM (Distributed Real-Time Ever Available Microcomputing) engine was formulated and its first prototype, DREAM kernel v1.0, was implemented on a PC LAN equipped with Intel 80486 processors, DOS-BIOS device drivers, the Packet Ethernet driver, and an interprocess multicast communication manager called the HU data field subsystem [Mor93, Kim95]. Services of the DREAM kernel including process management services can be obtained from within a C++ program (representing an implementation of an RTO.k object) via calls for DREAM library routines.

3.4 Interactions among objects

In some models, only the blocking type of service call is facilitated although the developers probably considered the issue of allowing non-blocking service calls a minor issue. In the MO2 model [Att91], the non-blocking service call is facilitated. The client-transfer call mechanism in the RTO.k object model is not available in any other model. The need for allowing client-transfer calls in the RTO.k model arose mainly due to the adoption of the basic concurrency constraint and the approach of design-time guarantee of timely services.

3.5 Maximum validity duration (MVD)

The specification and run-time checking of maximum validity durations were not adopted in any model but the RTO.k model. The usefulness of this facility is felt normally during the program/system validation. It should also be noticeable during the development of fault-tolerant hard-real-time systems.

3.6 Main differences between the RTO model in RTC++ and the RTO.k object model

Overall the object model in RTC++ [Ish92] and the MO2 model [Att91] are closer in nature to the RTO.k model than other models are. Since the differences between the RTO.k model and the MO2 model have been pointed out clearly in the above discussion, the RTC++ approach is reviewed in more detail here and compared against the RTO.k object model.

In RTC++, an RTO is declared as an active object and contains one or more locally possessed threads called *master threads* and incorporates specifications of timing constraints imposed on object methods and individual statements. It also defines a finite set of *slave threads* responsible for executing object methods called by clients. Each object is assigned a fixed priority and the priority of a client object is *inherited* by the server object during the execution of the method called by the client.

Main differences between the RTO model in RTC++ and the RTO.k model are the following:
(a) In RTC++, master threads, which are counterparts for the SpM's (spontaneous methods) in the RTO.k object, are not clearly separated in their roles from SvM's to the extent that SpM's in the RTO.k object are separated from SvM's. For example, SvM's in RTC++ cannot directly request executions of master threads but instead can perform by themselves all computing actions which would be done by SpM's in an RTO.k object.
(b) The basic concurrency constraint (BCC) can not be adopted in RTC++ since the approach of assigning fixed priorities to RTO's and the priority inheritance approach were adopted. Therefore, master threads cannot have higher priorities than SvM's in RTC++, which is the opposite of the BCC approach in the RTO.k model.
(c) No priorities are assigned to RTO.k objects. How to order competing accesses by object methods in execution for the same portion of the ODS is left to the execution engine which should utilize in its ordering decision the current information on time constraints associated with the competing methods.

4. Potential roles of the RTO.k structuring in engineering of real-time systems

Potential useful roles of the RTO.k model in engineering of complex real-time DCS's are now briefly discussed.

4.1 Uniform structuring of control computer systems and application environment specifications / simulators

The RTO.k object supports an interesting style of simulation. Suppose the application environment chosen is a sky+land+sea segment of interest, called the "theater", and any moving objects in that theater including ships and airplanes, etc. This environment can be represented and simulated by the RTO.k object in Figure 2.

The ODS in this RTO.k object contains state representations of the airplanes, the ships, and the theater space. This single RTO.k representation can be expanded into a representation in the form of a network of RTO.k objects, each representing an airplane or ship.

Each TT-method, when executed, updates a variable-set in the ODS representing the state of some physical object (i.e., airplane, ship) to reflect the current state of the physical object. Ideally the TT-methods should be *activated continuously* and each of their executions be *completed instantly*. This is fine when the object is used merely as a description rather than an executable program. However, such an execution engine for the RTO.k object cannot exist and thus we must adopt the less precise version of the model in which the time domain is a discrete domain, as an executable simulation model. That is, the limited power of the simulation engine dictates the activation frequency of any TT-method to be no more than once per every simulator clock tick while allowing each execution to be completed before or by the time of the following activation of the same method. Therefore, TT-methods are the mechanisms for simulating continuous state changes that occur naturally in the environment objects. The natural parallelism that exists among the environment objects is precisely represented by use of multiple TT-methods which may be activated simultaneously. In general, the accuracy of an RTO.k object structured simulation of the environment is a direct function of the activation frequencies of TT-methods.

Therefore, the RTO.k object model is an effective mechanism not only for variable-degree abstraction of real-time DCS's under design but also for variable-accuracy simulation of the application environments. Structures and notations used will be of the same kind in both control system design and environment simulation. The combination of a control system design and an environment simulator takes the form of a network of RTO.k objects. A specification and implementation experiment that involved an application of this RTO.k based uniform structuring scheme to both the development of a defense system and that of an environment simulator was conducted recently [Kim94b]. This involved the use of the DREAM kernel (v1.0) and concurrent programming in C++. This experiment confirmed the effectiveness of the uniform structuring approach and reinforced our belief that the RTO.k model offered an efficient and rigorous way to develop complex real-time systems.

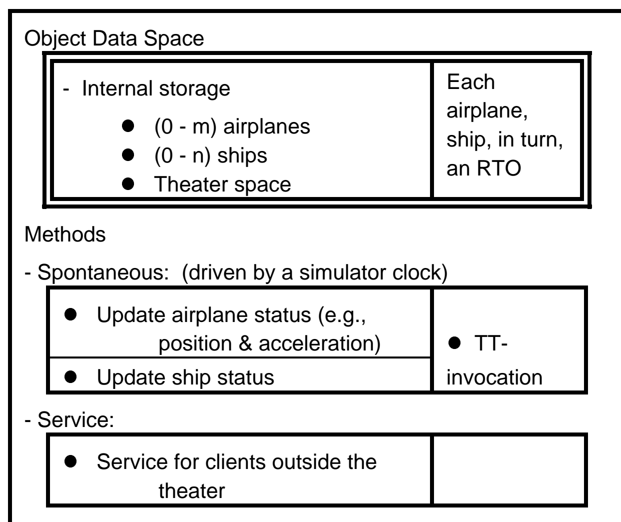

Figure 2. An RTO.k structured simulation model

We believe that the uniform structuring and accurate representation capabilities of the RTO.k object model can have positive consequences in all major phases of the real-time system engineering cycle such as the following :

(1) Requirement specification,
(2) High-level design,
(3) Stepwise refinement of a design to an implementation,
(4) Validation, and
(5) Maintenance.

Attempts to validate this hypothesis are considered highly meaningful subjects for future research.

4.2 Globally optimal resource allocation

We believe that the RTO.k structured specification of the requirements imposed on control computer systems provides much help in taking accurate global views of the temporal aspects of distributed cooperative computations. With such global views, systematic approaches to global resource allocation may become feasible. Some discussions on the issues to be resolved in realizing globally optimal resource allocation are given in [Kim94a].

5. Conclusion

We feel that the RTO.k object model offers some promises in achieving the *uniformity* and flexible and unrestricted degrees of *accuracy* in the representation of both application environments and control computer system designs evolving during the system development cycle. Realization of the full potential will require a great deal of future research of both analytical and experimental nature.

Acknowledgment: The research work reported here was supported in part by US Navy, NSWC Dahlgren Division under Contract No. N60921-92-C-0204, in part by the University of California MICRO Program under Grant No. 93-080, and in part by Hitachi, Ltd.

References

[Att91] Attoui, A. and Schneider, M., "An Object Oriented Model for Parallel and Reactive Systems", Proc. IEEE CS 12th Real-Time Systems Symposium, 1991, pp. 84-93.

[Bih89] Bihari, T., Gopinath, P., and Schwan, K., "Object-Oriented Design of Real-Time Software", Proc. IEEE CS 10th Real-Time Systems Symp., 1989, pp.194-201.

[Boo91] Booch, G., 'Object-Oriented Design', Benjamin Cummings, CA, 1991.

[Cha90] Champlain, M., “Synapse: A Small and Expressive Object-based Real-time Programming Language”, SIGPLAN Notices, Vol. 25, No. 5, 1990, pp. 124-134.

[Dah72] Dahl, O.J., “Hierarchical Program Structuring”, in Dahl, Dijkstra, & Hoare eds., ‘Structured Programming’, Aca. Press, NY, 1972.

[Her92] Hernandez, J. and Sanchez, J. A., “RT - MODULA2: An embedded in MODULA2 Language for writing Concurrent and Real Time programs”, ACM SIGPLAN Notices, Vol. 27, No. 2, February 1992, pp. 26-36.

[Ish90] Ishikawa, Y., Tokuda, H., and Mercer, C. W., “Object-Oriented Real-Time Language Design: Constructs for Timing Constraints”, Proc. ECOOP/OOPSLA ‘90, October 1990, pp. 289-298.

[Kim93] Kim, K.H. and Bacellar, L.F., “A Real-Time Object Model: A Step toward an Integrated Methodology for Engineering Complex Dependable Systems”, Proc. 1993 Complex System Engineering Synthesis and Assessment Technology Workshop, US Navy NSWC, July 1993, pp.56-64.

[Kim94a] Kim, K.H. et al., “A Methodology Framework for Optimal Design of Real-Time Dependable Computer Systems”, Proc. CSESAW ‘94 (1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop), US Navy NSWC, Dahlgren Div., July 1994, pp.251-259.

[Kim94b] Kim, K.H. and Kopetz, H., “A Real-Time Object Model RTO.k and an Experimental Investigation of Its Potentials”, Proc. 1994 IEEE Computer Society’s Computer Software and Applications Conf. (COMPSAC), Nov. 1994, Taipei, pp.392-402.

[Kim95] Kim, K.H., Mori, K., and Nakanishi, H., “Realization of Autonomous Decentralized Computing with the RTO.k Object Structuring Scheme and the HU-DF Inter-Process-Group Communication Scheme”, to appear in Proc. 1995 IEEE CS Int’l Symp. on Autonomous Decentralized Systems (ISADS), April 1995, Phoenix.

[Kop88] Kopetz, H. and Kim, K.H., “Consistency Constraints in Distributed Real Time Systems”, in M.G. Rodd and T.L. d’Epinay eds., ‘Distributed Computer Control Systems 1988’, Pergamon Press, 1989, pp.29-34.

[Kop90] Kopetz, H. and Kim, K.H., “Temporal Uncertainties in Interactions among Real-Time Objects”, Proc. IEEE CS 9th Symp. on Reliable Distributed Systems, Oct. 1990, pp.165-174.

[Mer90] Mercer, C.W. and Tokuda, H., “The ARTS Real-Time Object Model”, Proc. IEEE CS 11th Real-Time Systems Symposium, 1990, pp. 2-10.

[Mor93] Mori, K., “Autonomous Decentralized Systems: Concept, Data Field Architecture, and Future Trends”, Proc. Int’l Symp. on Autonomous Decentralized Systems (ISADS 93), Mar. 1993, Kawasaki, Japan, pp. 28-34.

[Rum91] Rumbaugh, J. et al., ‘Object-Oriented Modeling and Design’, Prentice Hall, New Jersey, 1991.

[Shi91] Shrivastava, S.K. and Waterworth, A., “Using Objects and Actions to provide Fault Tolerance in Distributed, Real-Time Applications”, Proc. IEEE CS 12th Real-Time Systems Symposium, 1991, pp.276-285.

[Tak92] Takashio, K., and Tokoro, M., “DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems”, Proc. OOPSLA, 1992, pp. 276-294.

[Wol91] Wolfe, V., Davidson, S., and Lee, I., “RTC: Language Support For Real-Time Concurrency”, Proc. IEEE CS 12th Real-Time Systems Symposium, 1991, pp. 43-52.