

# Principles of Constructing a Timeliness-Guaranteed Kernel and Time-triggered Message-triggered Object Support Mechanisms

K. H. (Kane) Kim \* and Chittur Subbaraman

Dept. of Electrical & Computer Engineering  
University of California  
Irvine, CA, 92697, U.S.A.  
kane@ece.uci.edu (\*: contact author)

**Abstract:** One of the major components required for the construction of future complex real-time computer systems (RTCS's) needed in safety-critical applications is a *timeliness-guaranteed operating system*. A model of an operating system kernel called the DREAM kernel that can support both conventional real-time processes and new-style real-time objects has been formulated recently. The key emphasis in formulating the DREAM kernel was in the realization of guaranteed timely responses to the service requests from supported client programs. This paper presents a summary of the main structuring principles that were exploited to realize guaranteed timely service capabilities while maintaining the modular and easily expandable structure in the DREAM kernel. Implementation of real-time processes and real-time objects involves designing various calls to DREAM kernel services. A library of functions, called the DREAM library and providing user-friendly interfaces to the DREAM kernel, has been developed in the authors' laboratory. This library consists of a collection of specific C++ classes. A 32-bit prototype version of the DREAM kernel, version D3.0, encapsulated by the DREAM library that supports well-structured real-time application programming in C++ has been implemented. This prototype version has been used in efficient development of several real-time object structured applications including a non-trivial defense C<sup>3</sup> system, a steel factory control system, and an advanced traffic management system.

**Keywords:** real-time operating system, object-oriented real-time system, time-triggered message-triggered object, real-time programming, DREAM kernel

## 1. Introduction

Both the customers and designers of large-scale complex real-time computer systems (RTCS's) are becoming increasingly anxious to see significant improvements in the reliability, expandability, and maintainability of the systems produced. Output actions taken by such systems are often subject to hard deadlines. Moreover, such systems should be structured in a modular form to make their maintenance and expansion to be manageable. In order to achieve significant improvements from the present state in the design efficiency and the system reliability attained, we believe that it will be most

rewarding to establish the following types of technologies for building complex real-time systems:

- (1) *General-form design style*: Future real-time computing must be realized in the form of a generalization of the non-real-time computing, rather than in a form looking like an esoteric specialization.
- (2) *Design-time guarantee of timely service capabilities of subsystems*: To meet the demands of the general public on the assured reliability of future RTCS's in safety-critical applications, there does not appear to be any adequate way but to require the system engineer to produce design-time guarantees for timely service capabilities of various subsystems (which will take the form of objects in object-oriented system designs).

The motivating factors behind these paradigms which may collectively be called the GG (General-form timeliness-Guaranteed) design are the recent significant advances in hardware economy and component reliability which have been fueling the expansion of the real-time computing application field.

An essential building-block for the realization of the idealistic GG design is a *timeliness-guaranteed operating system* which together with the hardware platform forms an *execution engine* that provides guaranteed timely services to concurrent and distributed real-time application software. If the operating system does not possess such guaranteed timely service capabilities, then the timely service capabilities of real-time applications cannot be ensured. Existing commercial operating systems provide insufficient capabilities for either supporting a general-form design style or providing guaranteed timely services to application software.

The first co-author recently formulated a model of an operating system kernel which can support real-time processes with guaranteed timely services. The model, named the DREAM (Distributed Real-time Ever Available Micro-computing) kernel, was born in the course of an attempt to develop an execution engine that supports a new real-time object-oriented (OO-) structuring approach. The new object structuring approach is called the TMO structuring scheme [Kim94a, Kim94b] and it was devised to facilitate the GG design. In principle, to realize the idealistic GG design a powerful structuring scheme capable of dealing with all practically useful real-time and non-real-time computing requirements must be

established. In the last several years, there has been a growing trend of research activities aimed for extending the conventional OO-structuring approaches to support RTCS design more effectively [Att91, Bih89, Ish92, Kim94b, Kop90]. However, most of those works have not been aimed for supporting the design-time guarantee of timely service capabilities of objects, which is one of the fundamental requirements of the GG design. Therefore, the TMO structuring scheme, though an extension of the conventional OO-structuring approaches has several unique features of fundamental nature that will be reviewed later in Section 3.1.

As an execution engine for TMO's, the DREAM kernel was designed to enable flexible linking between TMO's and various hardware structures. The kernel enables this by supporting

- (1) real-time processes subject to various activation and synchronization requirements,
- (2) shared data structure monitors called concurrent-read-&-exclusive-write (CREW) monitors, and
- (3) real-time multicast logical (RML-) channels called data field channels (DFC's).

These three basic components fully represent the general facilities for concurrent and distributed program structuring. Programs composed of these three components are called PCD (process-CREW-DFC) programs. In fact, the DREAM kernel can support both process-structured real-time application software (i.e., PCD programs) and TMO structured application software. Components of a TMO are treated as special types of PCD components. The key emphasis in formulating the DREAM kernel was in realization of guaranteed timely service capabilities with minimal loss of hardware utilization. The DREAM kernel can thus be viewed as a model of a general-purpose timeliness-guaranteed OS kernel.

In addition, a library of functions that eases not only PCD programming but also TMO programming in C++ have been designed. This library called the DREAM library is a collection of several C++ classes, each class functioning as an interface between a PCD or TMO program and a specific DREAM kernel service call (KSC) group. The DREAM library hides various details of parameter passing between application and the DREAM kernel and thus offers a user-friendly interface to the application programmer. An implemented prototype of the DREAM kernel encapsulated by the DREAM library supports not only well-structured PCD programming but also efficient programming of TMO's in C++.

One of the major goals of this paper is to provide an overview of the main structuring principles that were exploited to realize guaranteed timely service capabilities while maintaining the modular and easily expandable structure in the DREAM kernel. Among the main principles exploited is that of structuring multiple layers of "time-leasing" machines, each of which guarantees a

certain amount of machine time being available to upper-layer machines.

Several prototype implementations of the DREAM kernel have been produced in the authors' laboratory to run on networks of PC's connected by Ethernet. Recently, a 32-bit version of the DREAM kernel (version D3.0) encapsulated by the DREAM library has been implemented. This prototype version has been used to execute several TMO structured application systems including a non-trivial defense C<sup>3</sup> application, a steel factory control system, and an advanced traffic management system.

The paper starts in Section 2 with a discussion on the major architectural features of the DREAM kernel that enable guaranteed timely services to PCD programs. The entire DREAM kernel is then discussed in Section 3 and this section begins with a review of the essence of the TMO structuring scheme. The remainder of the section focuses mainly on the part of the DREAM kernel that is directly related to supporting TMO's. Section 4 discusses the major features of the DREAM library and Section 5 discusses the prototype implementations. The paper concludes in Section 6 with discussions on the issues to be resolved via future research.

## 2. Major architectural features of the DREAM kernel

This section focuses on the main structuring principles that were exploited to realize the DREAM kernel's guaranteed timely service capabilities.

### 2.1 Basic components of process-structured real-time concurrent and distributed programs

The DREAM kernel functions as a *process execution engine* by supporting the following three types of concurrent and distributed program components:

- (1) Processes: These are real-time processes, also called *application processes* (AP's) in this paper, with various synchronization and activation requirements. In reality, some of these processes may play the roles of system management processes, e.g., processes managing I/O. The processes may frequently impose deadlines on the kernel for the execution of their next computation-segments.
- (2) CREW (concurrent-read-&-exclusive-write) monitors: The CREW monitor is a shared data structure monitor and an extension of the *monitor* defined in [Bri77] in that the former possesses the *readers-writers* semantics (i.e., concurrent-read-&-exclusive-write semantics) instead of the exclusive-read-&-exclusive-write semantics associated with the latter.
- (3) Data field channels (DFC's) in the HU-DF scheme: The *HU-DF (HU data field) scheme* for inter-process-group communication [Kim97b] is an extension of the original *data field* scheme developed by Mori and other researchers in Hitachi, Ltd. [Mor93]. The essence of the

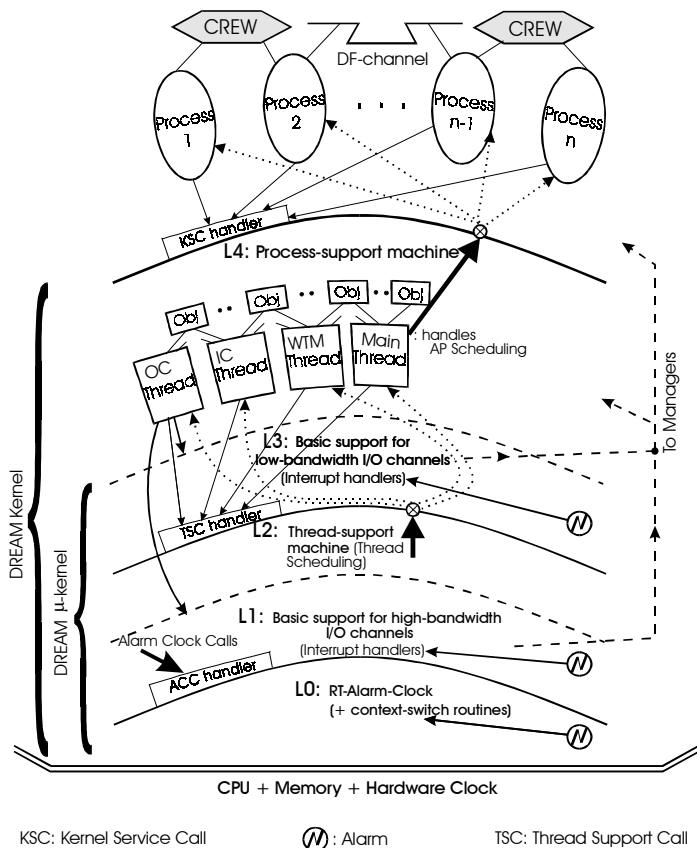


Figure 1. The architecture of the DREAM kernel

data field scheme is to facilitate dynamic creation of *logical multicast channels* called data field channels (DFC's) and dynamic connection of processes to the DFC's in such a way that the idiosyncrasies of the physical communication networks are transparent to the process designer. If the physical communication facility has the broadcast capability, then a logical multicast channel is facilitated by making all processing nodes using the channel to broadcast (through the physical communication facility) messages with the headers containing the ID of the channel called the content code. The processing nodes connected to the logical channel can see all the messages coming through the physical broadcast facility but will pay attention only to those messages containing relevant content codes. The HU-DF scheme differs from the original data field scheme in that the former allows dynamic flexible connection of processes to the logical channels and supports not only conventional *event messages* but also *state messages* which are based on the distributed replicated memory semantics (a variation of the state message semantics in [Kop89]).

The general facilities for concurrent and distributed program structuring are fully represented by these three basic components (process, CREW and DFC's). Therefore, any operating system kernel that supports these three components along with various I/O operations can be viewed as a general purpose kernel.

## 2.2 Five-layer structure and the principle of time-leasing machine layering

The layered architecture of the DREAM kernel is depicted in Figure 1. As shown, the DREAM kernel adopts a unique approach for the layering of its components. The key to the realization of the kernel's guaranteed timely service capabilities is this special layering approach. The kernel consists of five layers in total and the bottom four of the five layers constitute the DREAM micro-kernel. Whereas the DREAM kernel can be viewed as a *process execution engine*, the DREAM micro-kernel can be viewed as a *kernel-thread execution engine*.

The kernel-thread, or thread for short, is an active concurrency unit operating inside the DREAM kernel. They are used to exploit parallelism inside the kernel. The set of threads is fixed at the operating system loading time. All the threads share the same address space. Of the four basic threads, three are periodic threads. The fourth one called the *Main Thread (MT)* is executed whenever none of the other three threads are in execution. The MT is responsible for selecting the next process to run and causing the process to run within the time periods allocated to itself (MT). Once a thread is chosen to run, it can run for one time unit called thread-time-slice. If a periodic thread chosen to run does not need the full thread-time-slice, then it can "donate" the remaining portion of the time-slice to MT or just burn it by idling, depending upon the size of the remaining portion. The MT is thus "semi-periodic" in the sense that it becomes periodic when all other periodic threads use up their time-slices fully. Therefore, unlike the process scheduling, the thread scheduling is a simple low-overhead operation. We submit that restricting all kernel-threads except MT to be of this type only, i.e., periodic threads using one thread-time-slice at a time, is a well justified approach for making the analysis of the worst-case response time of each thread to be simple without much sacrificing the hardware utilization.

The special layering depicted in Figure 1 is based on the organizational principle called the time-leasing machine layering, which is of fundamental nature with respect to obtaining kernels with guaranteed timely service capabilities. Under this principle, the bottom layer, L0, owns the full power of the hardware machine. So, L0 uses the hardware machine at its own will. L0 contains a manager of a *real-time alarm clock* that supplies the current time upon receiving a request and also provides the "wake-up call" service. The remainder of the hardware machine time after L0's use of the hardware machine, is "leased" to the next upper layer, L1. L1 then uses a portion of the hardware machine time it receives from L0 (i.e., the machine time that is not used by L0). Similarly, L2 uses a portion of the hardware machine time that is not used by either L0 or L1. This *time leasing* relationship is applied *recursively* up to L4.

The upper four layers in the DREAM kernel contain the following components:

- (1) L1 contains basic support functions for high-bandwidth I/O such as the LAN interface;
- (2) L2 contains the thread scheduler which is activated upon expiration of a thread-time-slice or upon donation of an unfinished thread-time-slice by its current owner;
- (3) L3 contains basic support functions for low-bandwidth I/O such as serial character I/O;
- (4) L4 contains the process scheduler and other support functions for processes, CREW monitors, and DFC's. The process scheduler is activated upon expiration of a *process-time-slice* as well as at some other times. Here the size of the process-time-slice is usually an integral multiple of the size of the thread-time-slice.

Another important part of the time-leasing machine layering principle is

*to enforce that the amount of hardware machine time which is used by any layer during a basic response period be limited within a specific threshold.*

The *basic response period* here refers to the maximum interval between the instant at which a process calls for a kernel service and the instant at which the service is completed. The obvious purpose of adopting this principle is to guarantee a certain amount of machine time being available to each of the upper layers.

The amount of machine time used by L0 is quite stable and the upper bound on L0's use during a basic response period can be relatively easily calculated. The upper bound on L2's use or L4's use during a basic response period can also be calculated without much difficulty. However, the cases of L1 and L3 that must respond to interrupts from external sources are different. In principle, the LAN interface manager in L1 can experience bursts of interrupts generated by the LAN interface due to an incoming message burst. We may discover that it becomes impossible to guarantee timely delivery of any non-trivial services by L4 to processes under such conditions of L1. If so, *the frequency of interrupt generations by L1 must be controlled and set not to exceed a certain threshold.* This means that once the frequency reaches the threshold, any further interrupts by the interrupt source which causes the interrupt frequency bound to be violated are disabled. This in turn means that some incoming messages or signals will simply be lost. That is a price paid for guaranteeing timely delivery of all non-trivial services to AP's. On the other hand, if the LAN interface in the adopted machine configuration is incapable of generating interrupts at a frequency exceeding the threshold, then no dynamic enabling and disabling of such interrupts is necessary but it is still useful to understand the maximum possible frequency of generating interrupts because that information can be used in calculating the amount of machine time available to upper layers. We submit that this *dynamic control of*

*interrupt sources*, which is a part of the time-leasing machine layering principle, is a requirement that is inevitable in any timeliness-guaranteed operating system.

### 2.3 Kernel-threads

As mentioned in the preceding subsection (2.2), kernel-threads, except the semi-periodic MT, are periodic threads using one thread time-slice at a time. In the basic uniprocessor version of the DREAM kernel, there are four essential kernel-threads:

- (1) OCT (outgoing communication thread) which manages the sending of messages through the communication network,
- (2) ICT (incoming communication thread) which manages the distribution of messages coming through the communication network to the destination processes,
- (3) WTMT (watchdog-&-TMO-management thread) which manages the activation of TMO methods and checks if there are deadline violations, and
- (4) MT (main thread).

If the DREAM kernel were to operate as a process execution engine only, not as an execution engine for TMO's, then only a part of WTMT, i.e., only the watchdog timer service, is needed.

Data input from a high-bandwidth device is realized through a combined effort of the device, the interrupt handler in L1, and the ICT in L4. There is usually a modest-size buffer, say B1, into which a high-bandwidth device can pour data until it is filled. The device generates an interrupt when the buffer B1 is filled. In response to the interrupt, the interrupt handler in L1 moves the data in B1 into another larger buffer B2 accessible to both the interrupt handler and the ICT. The interrupt handler may insert many data items into B2 before the ICT becomes ready to access B2. Later when the ICT gets the next thread time-slice, it moves the data in B2 to the destination AP's.

One thing worth noting here is that if fast response activities specific to the application must be supported, then special application-specific threads can be introduced. However, the set of all kernel-threads must be fixed at the operating system loading time in order not to jeopardize the ability to guarantee timely service capabilities of the kernel at a reasonable performance level.

### 2.4 Atomic sections

The restriction of kernel-threads, except MT, to be periodic threads using one thread time-slice at a time enables both low-overhead scheduling of threads and easy analysis of such overhead. An additional measure taken to simplify the analysis of the worst-case response time of a kernel-thread is to resolve conflicts among kernel-threads in the DREAM kernel in accessing shared data without using locks for the shared data. Instead of using locks, a thread needing to access a shared data structure orders the thread support machine (in L2) not to disturb

the former, i.e., orders L2 to disable the thread switch so as not to let the machine power be taken away from the thread, until the thread notifies L2 that the "disturbance prohibition period" is over (and this obviously occurs when the thread finishes its access to the shared data structure). So, *even if a thread-time-slice expiration occurs during the disturbance prohibition period (i.e., while the thread is accessing the shared data), the thread support machine does not change the running thread.* Such a code-segment inside which a thread accesses shared data structures is called a thread-to-thread atomic section (TT-AS). The sending of a no-disturbance request to the thread support machine is thus an "Enter-TT-AS" operation, whereas, the cancellation of a no-disturbance request is an "Exit-TT-AS" operation. Again, the thread support machine supports neither data locks nor semaphores.

The TT-AS approach works only if the duration in which the thread executes the atomic section (AS) is much shorter than a thread time-slice. Generally this should be the case. Otherwise, either the thread time-slice was poorly chosen or threads and their shared data structures were not designed properly. This TT-AS approach is more efficient than the conventional data lock based approaches since kernel-threads except MT are periodic threads and frequent changes of the running thread can occur under the latter approaches but not under the former approach.

Also, in a manner analogous to the use of TT-AS's, AP's can use process-to-process atomic sections (PP-AS's). Therefore, synchronization between AP's can be realized via three different mechanisms: PP-AS, semaphore, and CREW monitor. Unlike in the case of kernel-threads, *the DREAM kernel also supports dynamic creation of AP's.*

## 2.5 Two-level scheduling

Scheduling of concurrent computation-units occurs in two different layers of the kernel since there are two types of concurrent computation-units, i.e., AP's and kernel-threads, supported by the DREAM kernel. *The process-time-slice should never be smaller than the thread-time-slice.* Normally, the process-time-slice should be multiple times as long as the thread-time-slice is.

The thread scheduler in L2 schedules the four basic threads, ICT, OCT, WTMT, and MT, for use of the hardware machine. Then the MT in L4, when it is executing its core (i.e., AP Scheduler, rather than an AP), schedules various AP's for use of the hardware machine. When an AP releases the control over the hardware machine voluntarily or under a mandate upon expiration of its process-time-slice, the control goes to the core (AP scheduler) of the MT.

The DREAM kernel provides a flexible platform to easily test out a variety of real-time AP scheduling policies. In the current prototype version of the DREAM

kernel, v.D3, the real-time AP scheduling policy incorporated is the *priority bracket (PB) scheduling* policy [Kim82]. The PB scheduling policy can be used to simulate a variety of other scheduling policies such as round robin, rate monotonic and other fixed priority scheduling policies, deadline-driven scheduling policies, etc. Nevertheless, the priority bracket scheduling policy is just one of many AP scheduling policies that can be incorporated into the DREAM kernel.

## 2.6 Low-overhead concurrency control

In Section 2.4 we discussed various forms of concurrency control mechanisms that are used in the DREAM kernel. Besides these concurrency control mechanisms, the DREAM kernel also employs a newly created low-overhead approach for mutually exclusive access to some critical variables by multiple layers.

According to the principle of time-leasing machine layering discussed in Section 2.2, when a layer  $L_i$  is in execution, its execution can be interrupted by any of the layers  $L_j$ , where  $0 < j < i \leq 4$  but not by any of the layers  $L_k$ , where  $0 < i < k \leq 4$ . Thus, when the layer L2 of the DREAM kernel is in execution, both the layers L0 and L1 are allowed to interrupt its execution, but the layers L3 and L4 are not. L2 contains the thread scheduler that is activated upon expiration of a thread-time-slice by the real-time alarm clock manager in L0 or upon donation of an unfinished thread-time-slice by its current owner. However, activation of the thread scheduler can be delayed longer than a thread-time-slice even if such a situation arises very rarely. This is because L2's execution can be prevented for quite some time due to a burst of message arrivals that incur L1 activities, TT-AS, etc. A delay of two thread-time-slices cannot occur if the threshold for the L1 interrupt frequency is set properly. Therefore, it is necessary for the alarm clock manager in L0 to keep track of the expiry of thread-time-slices while L2 remains prevented from execution. L0 needs to maintain a variable called *ThreadQuantumExpireCount* that must be incremented every time a thread-time-slice expires. The thread scheduler in L2 in turn must read this variable, discover how much delay has been incurred in its activation, and choose an appropriate thread for execution.

Unfortunately, the sharing of this critical variable between L0 and L2 poses the following problem. When the thread scheduler in L2 is trying to access the *ThreadQuantumExpireCount* variable, there is a possibility that its execution could be again interrupted heavily by the LAN interface manager in L1 due to the arrival of one or more bursts of messages at the LAN interface. This in turn will increase the execution time of the thread scheduler in L2. Thus, if the execution time of the thread scheduler increases by more than the value of a thread-time-slice, its execution will be interrupted by the alarm clock manager in L0 and while the thread scheduler remains in the middle of reading the shared variable *ThreadQuantumExpireCount*, the alarm clock manager

A segment of L0 Alarm Clock Manager (the following is executed whenever a thread-time-slice expires)	A segment of L2 Thread Scheduler (the following is executed every time L2 is activated)
<pre> switch (TQEC_Flag) {     case USING_TQEC1:         ThreadQuantumExpireCount_2 ++;         break;     case USING_TQEC2:         ThreadQuantumExpireCount_1++;         Break;     case USING_NONE:         ThreadQuantumExpireCount_1++;         break; } </pre>	<pre> TQEC_Flag = USING_TQEC1; ThreadQuantumExpireCount =     ThreadQuantumExpireCount_1; ThreadQuantumExpireCount_1 = 0;  TQEC_Flag = USING_TQEC2; ThreadQuantumExpireCount +=     ThreadQuantumExpireCount_2; ThreadQuantumExpireCount_2 = 0;  TQEC_Flag = USING_NONE; . . . // Use ThreadQuantumExpireCount to // choose an appropriate thread . . . </pre>

Figure 2. The low-overhead counter signaling protocol adopted in the DREAM kernel

could modify the variable, causing the thread scheduler to read an inconsistent value.

In order to prevent such an inconsistency, the "counter signaling" protocol shown in Figure 2 is adopted in the DREAM kernel. Here, instead of defining just one variable *ThreadQuantumExpireCount*, we define three variables *ThreadQuantumExpireCount\_1*, *ThreadQuantumExpireCount\_2*, and a flag variable called *TQEC\_Flag*. Before the thread scheduler in L2 accesses the variable *ThreadQuantumExpireCount\_1*, the *TQEC\_Flag* is set to *USING\_TQEC1*. When the *TQEC\_Flag* is in this state, the alarm clock manager in L0 can only modify the variable *ThreadQuantumExpireCount\_2*. Similarly, before the thread scheduler accesses the variable *ThreadQuantumExpireCount\_2*, the *TQEC\_Flag* is set to *USING\_TQEC2*, thus preventing concurrent access to the variable *ThreadQuantumExpireCount\_2* by the alarm clock manager. Note that *ThreadQuantumExpireCount* is set to the sum of *ThreadQuantumExpireCount\_1* and *ThreadQuantumExpireCount\_2* before it is used by the thread scheduler for choosing an appropriate thread. A formal proof of this protocol is omitted due to the space limit.

This form of concurrency control is much more efficient in this situation than conventional forms of concurrency control mechanisms (such as semaphores). Also, note that the probability of *ThreadQuantumExpireCount\_1* or *ThreadQuantumExpireCount\_2* changing during the execution of this signaling protocol in L2 is not very high, unless L1 heavily interrupts the execution of L2. Even if such a change occurs, it will be correctly accounted for by the next execution of the counter signaling protocol in L2.

A similar counter signaling protocol is also used between the thread scheduler in L2 and the AP scheduler in L4 for mutually exclusive access to a variable *ProcessQuantumExpireCount* that keeps track of the expiry of process-time-slices.

### 3. DREAM kernel support for the execution of TMO's

The remaining features of the DREAM kernel that do not belong to the process execution engine part, are the mechanisms directly related to supporting TMOs. After a brief review of the essence of the TMO structuring scheme in Section 3.1, those support mechanisms are discussed.

#### 3.1 The essence of the TMO structuring scheme

An initial abstract framework of the *TMO model*, formerly called the RTO.k object model, came out of an attempt to find a proper extension of the basic object model which is effective in structuring *hard-real-time* application systems. Based on the initial abstract framework formulated in late 1980's, a concrete syntactic structure and execution semantics was developed in recent years [Kim94a, Kim94b, Kim97a].

The basic structure of a TMO is depicted in Figure 3. It is an extension of the conventional basic object model(s) and two most important and unique extensions are the following:

- (a) Two clearly separated groups of methods:

For some methods of a TMO, a real-time clock serves as the mechanism for triggering the method executions as the clock reaches some values specified at

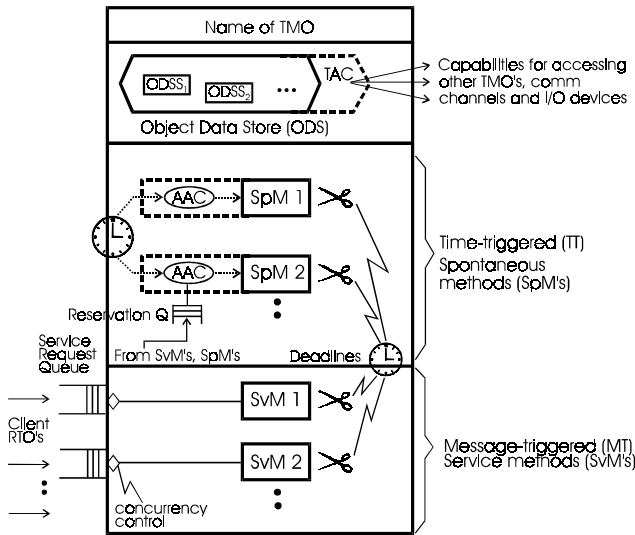


Figure 3. Structure of the TMO

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. The two types of methods in a TMO are different not only in the way their executions are triggered but also in that

"actions to be taken at real times which can be determined at the design time can appear only in SpM's".

Therefore, actions of the type "at constant-clock-value do S" or the type "sleep-until constant-clock-value" can appear only in SpM's. Incorporation of SpM's means introducing the potential for the following two new types of concurrent executions of object methods in addition to the potential for concurrent executions of SvM's that exist in conventional objects:

(Type I) Concurrency among SpM executions: This concurrency is specified in an implicit but natural manner, e.g., two SpM's designed to be triggered at 10 am.

(Type II) Concurrency between SpM executions and SvM executions.

(b) Basic concurrency constraint (BCC):

In order to dramatically reduce the designer's efforts in guaranteeing timely service capabilities of TMO's, the execution rule which prevents conflicts between SpM's and SvM's is incorporated. Basically, activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place. To be exact, when a message-triggered SvM is not free of conflict with an SpM in accessing the same portion of the object data store (ODS), execution of the former method (SvM) must not be allowed in a time zone earmarked for a TT-execution of the latter method (SpM). This restriction is called the basic concurrency

constraint (BCC). Therefore, executions of SpM's are not disturbed by SvM executions and triggering times of SpM's are fixed at the design time. If a statement of the type "at 10am do S" appears in an SpM, its timely execution can be easily assured.

The above two features make the TMO model clearly distinguished from other proposed real-time object models [Att91, Ish92]. In addition, the TMO contains a deadline for each output action and completion event of a method, which is a feature not found in the conventional basic object model but adopted in all discussions on a real-time extension of the basic object model.

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. An example of an AAC is

"for t = from 10am to 10:50am every 30min  
start-during (t, t+5min) finish-by t+10min"

which has the same effect as

```
{"start-during (10am, 10:05am) finish-by
start_time+10min",
"start-during (10:30am, 10:35am) finish-by
start_time+10min"}.
```

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 TMO requests future executions of a specific SpM.

An underlying design philosophy of the TMO model is that an RTCS will always take the form of a network of TMOs. The designer of each TMO 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. Before determining the deadline 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 deadline. Again, the BCC contributes to major reduction of these burdens imposed on the designer.

### 3.2 Overview of the approach for composing a TMO by using PCD program components

The DREAM kernel treats components of TMOs as special types of PCD components, i.e., special types of processes, CREW monitors, and DFC's. Figure 4 depicts the mapping relationship between components of a TMO and the corresponding components of an equivalent PCD-program.

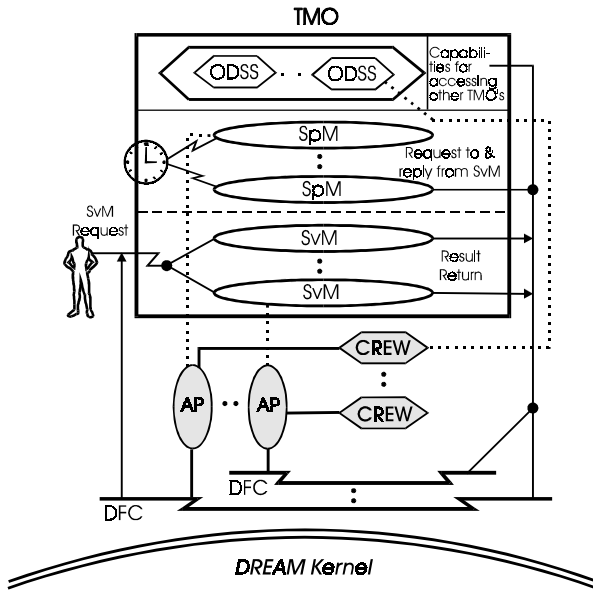


Figure 4. Mapping of a TMO to a PCD program

As shown in Figure 4,  
 (1) both SpM's and SvM's, are treated as application-specific program bodies of processes (called *method execution processes* or MEP's),  
 (2) access paths to SvM's as DFC's created by the SvM's,  
 (3) result-return paths to clients as DFC's created by the clients,  
 (4) ODS segments as special CREW monitors, and  
 (5) capabilities for accessing SvM's in other TMO's as DFC ID's.  
 Note that this way of utilizing DFC's facilitates the transparency of object locations.

A TMO method should be assigned to an MEP before the method can be executed. This assignment may be specified by the application programmer or may be left unspecified in which case the DREAM kernel will perform this function.

### 3.3 The watchdog and TMO management thread (WTMT) supporting TMO's

Of the kernel-threads shown earlier in Figure 1, the WTMT provides services needed mainly to support TMO's. It is a periodic kernel-thread responsible for:

- (1) timely activation and future reservation of SpM's,
- (2) activation of SvM's upon receiving the corresponding service requests,
- (3) enforcing the BCC, and
- (4) deadline checking for output actions and completion events of the methods and invoking an appropriate exception handler upon detection of a deadline violation.

A queue in layer L4, called the Reservation Queue (RvQ), holds the activation schedules of SpM's for the time window of next  $t$  time units. Reservations of SpM's

are done by inserting into the RvQ the earliest start time and the latest start time of each SpM which is to be activated during the next time period,  $t$ . The WTMT activates SpM's from the RvQ at their scheduled times by inserting the SpM's into the Ready AP Queue (RAPQ) and makes future reservations into the RvQ. Also upon receiving a service request message the WTMT will activate an SvM by inserting the SvM into the Ready AP Queue only if there is no possibility for the SvM to run into an ODS-conflict with any SpM. If an SpM which can run into an ODS-conflict with a SvM requested by an external client object is to be activated in  $d$  time units where  $d$  is less than the worst-case execution time of the SvM, the WTMT will delay the activation of the SvM.

Use of the WTMT for supporting TMO methods is an approach striking a good balance between the response time and the overhead, especially in comparison to the possibilities of using a dedicated process for similar functions or using the real-time alarm clock in L0 for some parts of the functions (e.g., TT-activation of an SpM).

### 3.4 Communication among TMO's

The HU-DF scheme has been adapted in the DREAM kernel to support communication among TMO's. One of the principal advantages of using the HU-DF scheme for inter-TMO method communication is that the scheme aids in enhancing the object autonomy. Event message channels are the major mechanisms for interconnecting client and server TMO's. The major direct benefit is the *transparency of object locations* which in turn leads to the enhanced relocation autonomy of the objects. The three different inter-TMO communication schemes incorporated into the DREAM kernel include communication via SvM calls, multicast-type communication, and communication via state messages.

### 3.5 Clock synchronization

The DREAM kernel supports the execution of applications structured either as real-time processes or as TMO's that are distributed among various nodes within a local area network. A global time with a sufficient precision must be maintained in the distributed system so that the distributed real-time applications can perform at an acceptable level. The DREAM kernel uses a fault-tolerant master-based clock synchronization scheme to maintain a global time. Our measurements with the DREAM kernel executing on a LAN of four i486, 66MHz PC's shows that this scheme provides a global time with a precision of about 2 msec. Attractive features of this scheme include low execution overhead and a small number of message exchanges between nodes.

Each node in the system is assigned a unique ID and the node with the smallest ID is initially chosen as the master node. The master node periodically broadcasts the local clock value to the network. The LAN interface manager in L1 executing in the slave nodes receives this

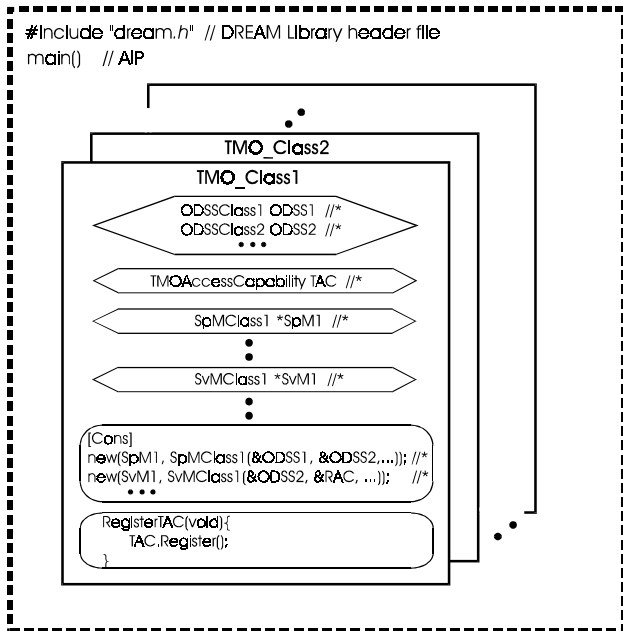


Figure 5. The definition of a TMO class

value. The received value is then corrected appropriately by the LAN interface manager to account for the message transmission time from the master node to the local host node, and then saved. The LAN interface manager also compares the local clock value maintained by the alarm clock manager in L0 and the corrected master clock value and decides on the rate at which the local clock has to be sped up or slowed down. It then issues an appropriate command to the periodically executing WTMT. When the WTMT is activated subsequently, it will discover a pending command from the LAN interface manager. It will then request the alarm clock manager in L0 to modify the interrupt generation frequency of the hardware timer appropriately. The alarm clock manager honors this request by issuing a command to the interrupt controller. Note that neither the LAN interface manager nor WTMT must request the alarm clock manager to simply set the local clock value to the corrected master clock value because such a modification may make the local clock miss one or more ticks or repeat the same ticks twice, both of which are undesirable.

The master-based clock synchronization scheme described above is vulnerable to failures in the master node. So, a master election scheme was incorporated into the above clock synchronization scheme.

#### 4. DREAM library support for PCD and TMO programming

Application programmers may construct each application program as a collection of C++ objects and the main function that is treated as the body of the special process called the Application Initial Process (AIP). The

AIP is the starting point of the entire application and is responsible for initializing the various child processes to run on the host node. The application program may request the services of the DREAM kernel during its execution by making kernel service calls (KSC's) to the DREAM kernel. An interface to the DREAM kernel that is friendly to the PCD as well as TMO programmers has been constructed. This interface is called the DREAM library. An early version of this library was discussed in [Kim96]. The DREAM library is a collection of several C++ classes, each class functioning as an interface between the application program and a specific KSC group.

The application program begins execution from an Application Initial Process (AIP). The C++ TMO programmer typically defines an TMO class as a C++ class and creates an instance of the TMO class inside the AIP. Figure 5 shows such TMO classes defined as C++ classes. In this figure, the *TMO\_Class1* represents one TMO class. It consists of

- (1) ODSS's created as instances of user-defined *ODSSClass*'s;
- (2) an instance of a user-defined *TMOAccessCapabilityClass*: Client TMO's need to obtain capabilities for accessing SvM's in server TMO's before requesting for service. The access capability will be provided to a client TMO in the form of the ID of the DFC that should be used for sending service request messages. The *TMOAccessCapabilityClass* facilitates a client TMO to obtain capabilities for accessing SvM's in server TMO's; and
- (3) instances of user-defined *SpMClass*'s and *SvMClass*'s: These objects represent the SpM's and SvM's of the owner TMO class.

The DREAM library provides a set of classes that can be inherited into the user-defined classes mentioned above. These DREAM library classes include the *BasicODSSClass*, the *BasicRTOAccessCapabilityClass*, the *BasicSpMClass*, and the *BasicSvMClass*.

#### 5. A prototype implementation of the DREAM kernel

Three different major prototype implementations of the DREAM kernel have been produced in the authors' laboratory to run on networks of PC's connected via Ethernet and equipped with Intel 80486 processors, DOS-BIOS device drivers, and the Packet Ethernet driver. The first two implementations were 16-bit prototypes and the most recent implementation (version D3.0) is the first 32-bit prototype. Several variations are also under development in a few cooperating organizations, including industry organizations and universities.

The DREAM kernel version D3.0 was used in experimental development of several real-time distributed application systems including a non-trivial defense C<sup>3</sup> system, a steel factory control system, and an advanced

traffic management system. The defense C<sup>3</sup> real-time distributed application software consists of nine different types of tailorable TMO's and runs on a network of three or more PC's. Some real-time fault tolerance capabilities were also implemented. During the development of these applications, we observed that the debugging efforts required were a relatively small fraction of the efforts needed for earlier implementations of similar but simpler applications by using conventional real-time process structured design methods. The application software consists of approximately 25,000 lines of C++ code. This experiment gave us a considerable amount of confidence in the soundness of the architecture of the DREAM kernel as well as the power of the TMO structuring scheme. The DREAM kernel, version D3.0, consists of approximately 20,000 lines of C++ code. The thread-time-slice used was 2 msec and the process-time-slice used was 16 msec.

## 6. Conclusion

The DREAM kernel can be extended in several major directions. The most obvious among them is to extend it to utilize multi-processor architectures and highly parallel machine architectures. Adapting the DREAM kernel to commercial micro-kernel environments is another meaningful direction to pursue. We plan to implement in the near future another version of the DREAM kernel that uses the Windows NT as its component. The interesting issue during such implementation will be at what time granularity we will be able to guarantee timely services. The DREAM library presents a convincing case of benefiting significantly in concurrent programming from the use of the advanced inheritance features of C++. Full realization of GG design will require investment of sizable efforts by industry to revise and extend their existing operating system products to possess guaranteed timely service capabilities and it is our hope that industry find some of the approaches and principles exploited in the DREAM kernel useful in such efforts.

**Acknowledgments:** The research work reported here was supported in part by the US Defense Advanced Research Project Agency under Contract N66001-97-C-8516 monitored by NRD, in part by the University of California's MICRO Program under Grant 96-169, in part by LG Electronics, and in part by the USAF Rome Lab under Contract F30602-96-C-0060.

## References

- [Att91] Attoui, A. and Schneider, M., "An Object Oriented Model for Parallel and Reactive Systems", *Proc. IEEE CS 12th Real-Time Systems Symp.*, 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.
- [Bri77] Brinch Hansen, P., "The Architecture of Concurrent Programs," *Prentice-Hall*, 1977.
- [Ish92] Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Computer*, October 1992, pp. 66-73.
- [Kim82] Kim, K.H., Abouelnaga, A., Heu, S., and Yang, S.M., "Process Scheduling and Prevention of Communication Deadlocks in an Experimental Microcomputer Network", *Proc. IEEE Computer Society's Real-Time Systems Symposium*, Dec. 1982, pp.124-132.
- [Kim94a] Kim, K.H. et. al, "Distinguishing Features and Potential Roles of the TMO Model", *Proc. 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Oct. '94, Dana Point, pp.36-45.
- [Kim94b] Kim, K.H. and Kopetz, H., "A Real-Time Object Model TMO and an Experimental Investigation of Its Potentials", *Proc. 1994 IEEE CS Computer Software and Applications Conf. (COMPSAC)*, Nov. 1994, Taipei, pp.392-402.
- [Kim96] Kim, K.H. et. al, "The DREAM Library Support for PCD and TMO Programming in C++", *Proc. 1996 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Feb. 1996, Laguna Beach, CA, pp. 59-68.
- [Kop89] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Wolfgang, S., Senft, C., and Zainlinger, R., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE Micro*, Feb. 1989, pp. 25-39.
- [Kim97a] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No. 8, August 1997, pp. 62-70.
- [Kim97b] Kim, K.H., and Subbaraman, C., "Interconnection Schemes for RTO.k Objects in Loosely Connected Real-Time Distributed Computer Systems", , *Proc. 21st IEEE Computer Software and Applications Conference (COMPSAC)*, Washington D.C, August 1997, pp 121-127.
- [Mor93] Mori, K., "Autonomous Decentralized Systems: Concept, Data Field Architecture, and Future Trends", *Proc. IEEE CS Int'l Symp. on Autonomous Decentralized Systems (ISADS 93)*, Mar. 1993, Kawasaki, Japan, pp. 28-34.