J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

REAL-TIME OBJECT-ORIENTED DISTRIBUTED COMPUTING

Real-time (*RT*) object-oriented (OO) distributed computing is a form of *RT* distributed computing realized with a distributed computer system (*DCS*) structured in the form of an object network. During the last two decades of the twentieth century, OO design approaches became a common practice in the development of non-real-time business data processing software owing to the modularity, reusability, and natural abstraction benefits offered by the OO approaches. On the other hand, OO structuring started to have a recognizable impact in *RT* computing near the end of the twentieth century. This is because the conventional OO structuring technology used in engineering of non-real-time business data processing software needed to be extended in major ways in order to be effective in *RT* software engineering. In other words, *RT* OO structuring is a major extension of conventional OO structuring.

RT OO structuring has the following important goals:

(1) *General-form design of RT computer systems.* It is highly desirable to realize *RT* computing in the form of a generalization of non-real-time computing, rather than as an esoteric specialization. With a properly established *RT* computer system design methodology, it should be possible to realize every practically useful non-RT computer system by simply filling the time-constraint specification part with unconstrained default values. Such a design becomes a reality only when a powerful structuring scheme capable of dealing with all practically useful *RT* and non-real-time computing requirements is established. *RT* OO structuring is an approach for meeting such requirements. Therefore, it is aimed at facilitating uniform structuring of hard *RT* computation, soft *RT* computation, and non-*RT* computation. Here, *hard RT computation* refers to computation that is subject to stringent timing constraints. The timing constraints are stringent in two respects. First, if they are violated, then the resulting damage to the application environment or customer is intolerably high. Second, the required timing precision of critical computing actions (e.g., sending braking commands to an automobile brake system or launch commands to a defense missile launcher) is very high (e.g., millisecond-level precision). On the other hand, the timing constraints imposed on *soft RT computation* are loose.

(2) *Design-time guarantee of timely service capabilities of subsystems.* To meet the demands of the general public for assured reliability of complex *RT* computer systems in safety-critical applications, there is currently no 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 OO system designs). Experience of practicing engineers indicates that testing alone is not sufficient to ensure the level of reliability of complex *RT* computer systems that customers are demanding. Even if verifying the full logical behavior of a sizable *RT* software may be impractical, verification of the important timing behavior can be practical and must be pursued especially when hard *RT* applications are dealt with. This approach is further supported by the fact that it is the timing behavior that presents the biggest difficulties to the system engineer relying on testing for assuring proper system behavior to a reasonable degree. The ease or difficulty of verifying the timing behavior can also be viewed as a measure of the *timing behavior predictability* of the given system

design. *RT* OO structuring is a major design approach aimed at enabling design-time guarantees of timely service capabilities of software subsystems structured as objects.

While designed to meet the fundamental goals just mentioned, *RT* OO structuring solves or at least reduces the following practical problems that existed in *RT* computing for decades through the end of the twentieth century.

(1) *Inefficient design and implementation.* As mentioned before, the conventional OO structuring technology had minimal impact on *RT* computer system engineering in contrast to its pervasive use in non-RT computer system engineering. This means that much of the capabilities existing in the vast business data processing software field has not been greatly utilized in the development of *RT* computer systems. It also means that the *RT* computer system engineering process used and the *RT* application software developed before the emergence of the *RT* OO structuring approaches took by and large peculiar forms unfamiliar to the vast mainstream software engineering community. The consequence was the poor economy of scale in *RT* computer system development and the relatively low reliability of the software products except in cases of small-scale simplistic phase-lock loop control types of applications. The general-form design style facilitated by *RT* OO structuring should solve this problem in major ways.
(2) *Low reliability of large-scale RT computer systems.* Various representation schemes, analysis and synthesis techniques, and tools have been developed for use in each phase of *RT* computer system engineering, such as specification, design, implementation, validation, and maintenance. By and large, these techniques have not evolved in sufficiently integrated forms up to near the end of the twentieth century. As a result, *RT* computer system engineering practices suffered from the following problems: (a) weak traceability among various system models used, (b) lack of rigor in requirements specification, and (c) lack of integration in design techniques. Moreover, experience has shown that *RT* distributed computing software is notoriously difficult to test. All these lead to the fact that the reliability of the large-scale *RT* software produced before the emergence of *RT* OO structuring approaches is not sufficiently high.

Fundamental Issues in *RT* OO Distributed Computing

Several fundamental issues in establishing *RT* OO distributed computing technology in a sound form are discussed in the following. One common underlying theme is that desirable types of *RT* distributed objects should not only facilitate modular and economic design of distributed systems and high-precision *RT* application systems but also be easily interconnected and integrated into an easily analyzable high-level function system. That is, a desirable technology should yield a high degree of *composability*.

Specification of Timing Constraints. Clear specification of timing constraints is a fundamental requirement in rigorous engineering of *RT* computer systems. Major issues in this area are

(1) Global time base
(2) Time-triggered (*TT*) action
(3) Separation of the absolute time domain from the relative time domain

Each of these issues will be discussed in some detail. In the rest of this article, an object that contains specification of timing constraints is called a *real-time (RT) object*.

Global Time Base. In order to effect cooperative distributed *RT* computing in an efficient manner, a global time base that supplies the time-of-the-day information to distributed processing nodes and distributed

objects in a *DCS* must be established. Quite a few ways for establishing global time bases of varying precision are known.

In any practical *RT* system design or programming language, the following features must be included:

(1) *Specification of time bases.* This includes specifying *UTC* (universal time coordinated), SST (the time elapsed since the system started), etc.
(2) *Global-time reference function.* This includes *now*, which returns the current time obtained from the global time base; *forever*, which is a time constant representing a practically infinite time interval; etc.

Time-Triggered (TT) Action. Specification of *TT* computations is a fundemental feature of *RT* programming that distinguishes *RT* programming from non-RT programming. The computation unit can be any one of the following:

(1) A simple statement such as an assignment statement with the right-side expression restricted to an arithmetic logical expression type involving neither a control flow expression nor a function call, an input/output (I/O) command statement, etc.
(2) Compound statement such as *if–then–else* statement, *while–do* statement, *case* statement, etc.
(3) Statement block
(4) Function and procedure
(5) Object method

Time-triggered actions associated with a computation unit may include a timely start of the computation unit, timely completion of the computation unit, and periodic execution. Therefore, in any practical *RT* system design or programming language, it is desirable to have the following type of a construct:

$$
\begin{array}{ll}
ab & \text{"timing specification begin"} \\
 & \text{for <time-var>} = from\text{<activation-time>} \\
 & \quad to\ \text{<deactivation-time>} \\
 & \quad [every < \text{period} >] \\
 & start - during \\
 & \quad (\text{<earliest-start-time>}, \\
 & \quad \text{<latest-start-time>}) \\
 & finish - by \quad < deadline > \\
ae & \text{"timing specification end"}
\end{array}
$$

For example, consider the following case:

$$
\begin{array}{l}
"for\ \mathrm{t} = from\ 10\mathrm{am}\ to\ 10:50\mathrm{am}\ every\ 30\mathrm{min} \\
start - during\ (\mathrm{t}, \mathrm{t} + 5\mathrm{min})\ finish - by\ \mathrm{t} + 10\mathrm{min}"
\end{array}
$$

This specifies: "The associated computation unit must be executed every 30 min starting at 10 A.M. until 10:50 A.M., and each execution must start at any time within the 5 min time window $(t, t + 5$ min) and must be completed by $t + 10$ min." So, it has the same effect as

$$
\begin{array}{l}
\{start - during\ (10\mathrm{am},\ 10{:}05\mathrm{am})\ finish - by\ \text{start-10:10am}", \\
"start - during\ (10{:}30\mathrm{am},\ 10{:}35\mathrm{am})\ finish - by\ \text{start-10:40am}"\}
\end{array}
$$

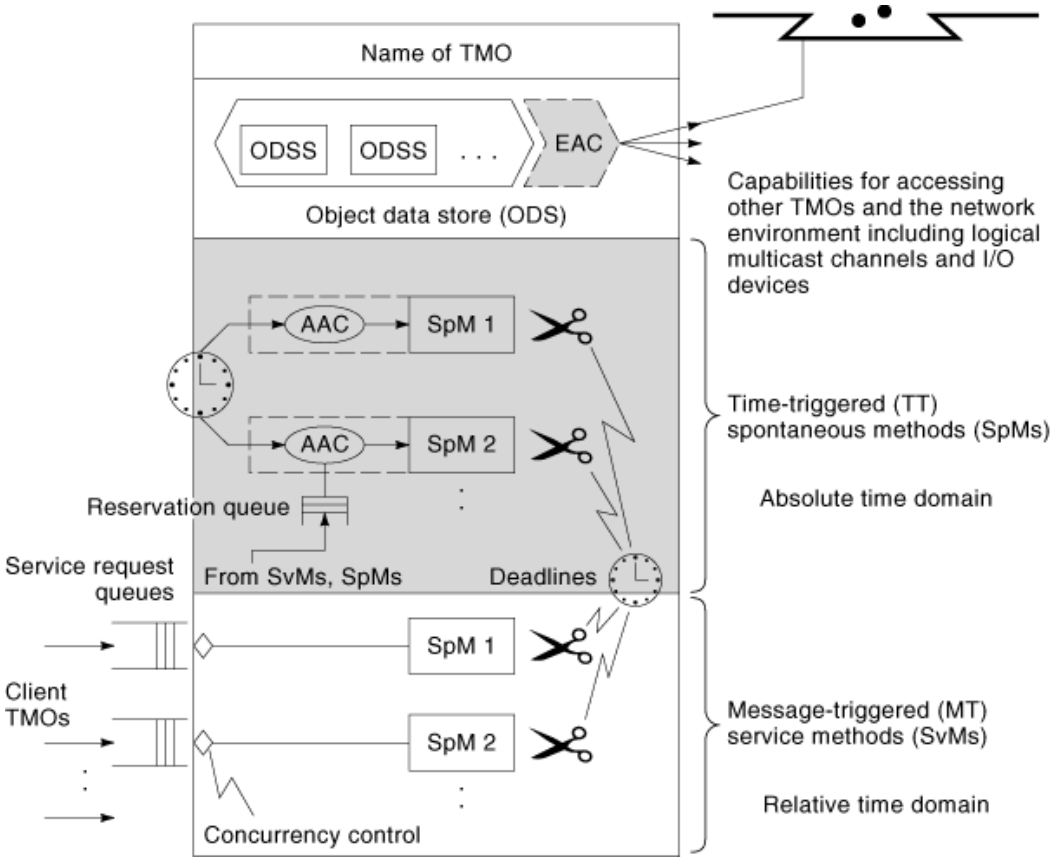

Fig. 1. Example RT-object structure: *TMO* (time-triggered message-triggered object).

Also, it is good to have the construct after time-const do *S* " in a *RT* programming language although the same effect can be achieved by using the above *for–start–finish* construct.

Of the five types of computation units mentioned before, the object method is the most frequently used unit for *TT* start and timely completion. A representative example of *RT* OO structuring approaches that support the design of *TT* methods is the *Time-Triggered Message-triggered Object* (*TMO*) structuring scheme. Figure 1 depicts the basic structure of the *TMO*. In this scheme, *TT* methods are also called *spontaneous methods* and they are "clearly separated" from the conventional methods that are invoked by object clients and called *service methods*. The reason for a clear separation of the two groups of methods will become clear later. Timing specifications associated with *TT* methods are called *autonomous activation conditions* (*AACs*).

In the *TMO* structuring scheme, there is also a provision for making the *AAC* section of a *TT* method 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 a service method within the same *TMO* requests future executions of a specific *TT* method with the parameters that indicate the times selected from the candidate triggering times.

Separation of the Absolute Time Domain from the Relative Time Domain. From the viewpoint of obtaining easily understandable and analyzable *RT* programs, it is also good to clearly separate the specification of the computation dealing with the *absolute time domain*, that is, the computation dependent on the time-of-day

information available from the global time base, from the specification of the computation dealing with the *relative time domain* only. In the case of the *TMO* structuring scheme depicted in Fig. 1, service methods deal with the relative time domain only, that is, they use only the elapsed intervals, since the method was started by an invocation message from an object client. This is natural, since the arrival time of a service request (that is, a message invoking a service method) from an object client cannot be predicted by the designer of the service method in general, especially when that designer is not the designer of the client object. Therefore, any use of the time-of-day information can be used within *TT* methods only. This means that computations of the type " at constant-global-time-value do *S* " or the type " after constant-global-time-value" can appear only in *TT* methods. The only exception is that an arithmetic logical expression consisting of *now* and global time constants may be used in a server method for the purpose of selecting candidate triggering times associated with *TT* methods. In fact, the rule adopted in the *TMO* scheme can be stated as: "actions to be taken at real times *that can be determined at the design time* can appear only in *TT* methods."

The discussion of the specification of *TT* actions, that is, computation dealing with the absolute time domain, was given in the preceding subsection. For specifying the computation dealing with the relative time domain only, the following language features are useful:

$$start - within \text{ duration} - 1 \; complete - within \text{ duration} - 2 do \; \mathbf{S;}$$
$$start - within \text{ duration} - 1 \; complete - within - after - start \text{ duration} - 2 \; do \; \mathbf{S;}$$

Concurrency Control. A real-time object may contain various types of concurrency. For example, in the case of the *TMO* depicted in Fig. 1, the following major types of concurrency may be found:

(1) Concurrency among *TT* method executions, that is, the concurrency specified in an implicit but natural manner (e.g., two *TT* methods designed to be triggered at 10 A.M.)
(2) Concurrency among service method executions
(3) Concurrency between *TT* method executions and service method executions

In order to maintain a high degree of timing behavior predictability in an *RT* object, concurrency within the object must be clearly understood and controlled.

A highly useful principle here is to first use the *basic concurrency constraint* (*BCC*) adopted in the *TMO* scheme. *BCC* prevents conflicts between *TT* methods and service methods in accessing a common part of the object data store (*ODS*). It contributes in an important way to reducing the designer's efforts in guaranteeing timely service capabilities of *TMO*s. 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 with a *TT* method in accessing the same portion of the *ODS*, execution of the service method must not be allowed in a time zone earmarked for a *TT* execution of the latter method. Therefore, *TT* methods are given higher priorities for execution over the service methods. Executions of *TT* methods are not disturbed by service method executions, and triggering times of *TT* methods are fixed at the design time. If a statement of the type " at 10am do *S* " appears in a *TT* method, its timely execution can be easily assured.

Another concurrency control approach useful, though not as essential as the *BCC*, in easing the design-time guaranteeing of timely services is to follow the *ordered isolation rule*. In order to describe this rule, the *TMO* structure will be considered again. The term *initiation time stamp* or *I–time stamp* is defined as follows. In the case of a service method execution, the I–time stamp is defined as the record of the time instant at which the execution engine initiated the service method execution after receiving the client request and ensuring that the service method execution can be initiated without violating the *BCC* and other execution rules. In the case of a *TT* method execution, the I–time stamp is defined as the record of the time instant at which the *TT*

method execution was initiated according to the timing specification of the *TT* method. Also, a segment of the *ODS*, called an *ODS* segment (ODSS), is a basic unit of data storage that can be reserved for exclusive access by a method of a *TMO*. The *ordered isolation rule* can be stated as follows:

(1) A method execution with an older I–time stamp must not be waiting for the release of an ODSS held by a method execution with a younger I–time stamp.
(2) A method execution must not be rolled back because of an ODSS conflict.

The ordered isolation rule can be illustrated by considering two service methods, SVM1 and SVM2, that need to access the same ODSS, $ODSS_7$, mutually exclusively during their execution. Assume that a client request arrived first for SVM1 and another client request arrived later for SVM2. The execution engine will initiate SVM1 first with an older I–time stamp and SVM2 later with a younger I–time stamp. Both SVM1 and SVM2 executions may proceed concurrently. Then under the ordered isolation rule, the SVM1 execution should never have to wait for entering $ODSS_7$ due to the SVM2 execution accessing $ODSS_7$ at the same time. Moreover, meeting such a condition by forcing the SVM2 execution out of $ODSS_7$ and rolling it back to its beginning (i.e., by "wounding" the SVM2 execution) is not allowed. Approaches that are less restrictive than the ordered isolation rule and yet ease the design-time guaranteeing of timely service capabilities of *RT* objects are certainly desirable but such approaches have not been established yet.

Note also that the approach of *pipelined execution of service methods* can be incorporated. When the service request rate for a service method becomes high, multiple instances of the same service method may be executed concurrently in a pipelined fashion to speed up the processing of client requests. The *RT* object designer must specify explicitly when such a pipelined execution of service methods should be performed.

Interaction among *RT* Objects and *RT* Message Communication.

Nonblocking Call. An underlying design philosophy of the *RT* OO distributed computing approaches is that every *RT DCS* will be designed in the form of a network of *RT* objects. *RT* objects interact via calls by client objects for service methods in server objects. The caller may be a *TT* method or a service method in the client object. In order to facilitate highly concurrent operations of client and server objects, *nonblocking* (sometimes called asynchronous) types of calls (i.e., service requests) in addition to the conventional *blocking* type of calls can be made to service methods. Therefore, the following two basic types of calls can be made to service methods in the server *TMO*.

(1) *Blocking call.* After calling a service method, the client waits until a result message is returned from the service method. The syntactic structure may be in the form of Obj-name. SvM-name(parameter-1, parameter-2, . . ., by deadline). 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 service method, the execution completion signal is returned to the client. If the result message or the execution completion signal from the server method does not arrive by the specified deadline, the execution engine for the client object invokes an appropriate exception handling function as it would when an arithmetic overflow occurs.
(2) *Nonblocking call.* After calling a service method, the client can proceed to follow-on steps (i.e., statements or instructions) and then at some point wait for a result message from the service method. 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 nonblocking call. When the client calls the service method, the client records a time stamp into a variable, say TS. The time stamp uniquely identifies this particular call for the service method as distinct from other (past or future) calls for the same service method 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 nonblocking call for the service method, not only the service method name but also the variable TS containing the time stamp associated with the subject call must be indicated. When a client makes multiple nonblocking calls for service methods before executing a "get-result" statement, the time stamp unambiguously indicates to the execution engine which nonblocking 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 nonblocking call thus creates concurrency between a client method (TT method or service method) and a service method in a server object and the concurrency 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 a service method. Such a client does not use a "get-result" statement.

RT Message Communication and Programmable Multicast Channels. Whether a service request is a blocking call or a nonblocking call, the request message and the result return message must be communicated with predictable delay bounds. Many protocols suitable for *RT* message communication over local area networks and wide area networks exist, for example, time-division multiplexed access (*TDMA*), token ring access, deterministic CSMA/CD, ATM, etc.

In addition to the interaction mode based on remote method invocations, distributed *RT* objects can use another interaction mode where messages may be exchanged over message channels explicitly specified as data members of involved objects. For example, logical multicast channels, *LMC1* and *LMC2*, can be declared as data members of each of the three remotely cooperating *RT* objects, *TMO1*, *TMO2*, and *TMO3*, during the design time. The compiler and the object execution engines running the three *RT* objects must then together facilitate the two channels and guarantee timely transmission of messages over those channels. Once TMO1 sends a message over *LMC1*, the message will be delivered to the *ODS* of each of the three *RT* objects. Certain methods in TMO2 and TMO3 can pick up the messages that came over *LMC1* into the ODSs of their host objects. In many applications, this interaction mode leads to better efficiency than the interaction mode based on remote method invocations does.

Transparency of RT-Object Locations. In most cases, the RT-object designer does not want to be burdened with the chore of learning and utilizing the information on the physical locations where other cooperative *RT* objects will be running. Therefore, the compiler and the object execution engine together must relieve such a designer of the concern about the physical locations of cooperating *RT* objects. Several approaches exist.

In 1990s, an industry consortium led by Object Management Group launched a movement to establish common object request broker architecture (*CORBA*), a common architectural framework for distributed computing across heterogeneous hardware platforms, operating systems, and internode communication protocols. *CORBA* is also a location-transparent language-independent interobject communication architecture. The core of *CORBA* is the object request broker (*ORB*), a mechanism that supports remote object method invocation and interobject communication. The *ORB* provides a high-level location-transparent abstraction for facilities

for communication among objects residing in different hosts. Another approach is to make object execution engines fully cooperate in searching for *RT* objects in a network domain.

Deadline Specification and Service Time Guarantee. The designer of each *RT* object provides a guarantee of timely service capabilities of the object by indicating the *guaranteed time-window for every output* produced by each service method in the specification of the service method advertised to the designers of potential client objects. Actually the guaranteed time-window associated with every output from every *TT* method also a part of the guarantee. Before determining the guaranteed time-window specification, the server object designer must convince himself or herself that with the *object execution engine* (hardware plus operating system) available, the server object can be implemented to always execute the service method such that the output action is performed within the time-window. Again, the *BCC* contributes to major reduction of these burdens imposed on the designer.

An output action of a service method may be one of the following:

(1) An updating of a portion of the ODS
(2) Sending a message to either another *RT* object (which may or may not be the client) or a device shared by multiple objects
(3) Placing a reservation into the *reservation queue* for a certain *TT* method that will in turn take its own output actions

The specification of each service method that is provided to the designers of potential client *RT* objects must contain at least the following:

(1) An *input specification* that consists of (a) the types of input parameters that the server object can accept and (b) the *maximum request acceptance rate*, that is, the maximum rate at which the server object can receive service requests from client objects
(2) An *output specification* that indicates the *output time-window* and the *nature of the output value* for every output produced by the service method

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, the server may return exception signals to the client objects. The system designer who checks an interconnection of *RT* objects can prevent such overflow occurrences through a careful analysis. The designer should ensure that the aggregate arrival rate of service requests at each server object does not exceed the maximum acceptance rate during any period of system operation. In order to satisfy greater service demands presented by the client objects, the system designer can increase the number of server objects or use more powerful execution engines in running server objects.

Before determining the output time-window specification, the server object designer must consider the following.

(1) The worst-case delay from the arrival of a service request from a client object to the initiation of the corresponding service method by the server object
(2) The worst-case execution time for the service method from its initiation to each of its output actions

On the other hand, a client *RT* object imposes a *deadline* on the server *RT* 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 maximum delay guaranteed by the designer of the server object for completion of the corresponding service method.

The specifications of the *TT* methods that may be executed on requests from the service method must also be provided to the designers of the client objects that may call the service method. The specification of such a *TT* method must contain at least the time triggering specification and the output specification. There is no input specification. The output specification indicates, for every output expected from the execution of the *TT* method, the time-window during which it will be produced and the nature of every value carried in the output action.

Inheritance. Inheritance is one of the major features of the OO design approach. Once a class that contains capabilities commonly required by a group of applications is created, each application in the group can be efficiently constructed in the form of a derived class that inherits all the capabilities of the base class and incorporates additional newly designed capabilities.

The inheritance aspect of the *RT* object has one more dimension than the conventional object, namely dealing with timing attributes. For example, when an *RT* class is derived from a base *RT* class, the *TT* initiations, deadlines, and other specifications of action timings inside the base *RT* class can all be inherited. If some of the timing specifications are not acceptable in a new application situation, then those can be replaced by new specifications in the derived class.

Design and Execution Tools

Execution Support Middleware. To facilitate *RT* OO structuring in the most cost-effective manner, it is essential to provide execution support mechanisms in well-established commercial software–hardware platforms compliant with industry standards. Execution support mechanisms for *RT* objects have been built as middleware running on popular commercial operating system kernels. Any operating system kernel with the following capabilities can support middleware supporting *RT* objects.

(1) High-precision time base.
(2) RT threads possessing the following characteristics: (a) the delay an *RT* thread experiences in accessing a resource should not exceed a predetermined tight bound, (b) it can enter a nonpreemptible mode at any time, and (c) when a kernel service call is issued inside a *RT* thread, the kernel service inherits the *RT* characteristics of the issuer thread.
(3) Efficient control of support processes, that is, long-life background support processes (often called daemons) for performing day-to-day management and housekeeping activities

Programming Language Tools. Language tools for RT-object programming need not take the form of completely new languages. Well-established basic OO programming languages such as C++ and JAVA can be used as a base language and by adding an appropriate library, a cost-effective language tool can be obtained. A few such tools have been constructed.

Aids for Analysis of the Worst-Case Service Time. The most urgently needed among various software engineering tools desired but unavailable at the time of writing are those for assisting the RT-object designer in the process of determining the response time (i.e., maximum delay for each output) to be guaranteed. Such tools must be capable of making good estimates for worst-case execution times of various segments of object methods. Such tools can also be useful in checking the *execution feasibility* of *RT* objects, for example, checking if a group of *RT* objects can be loaded onto a *DCS* node without introducing the possibility of any violation of the timing constraints associated with the *RT* objects.

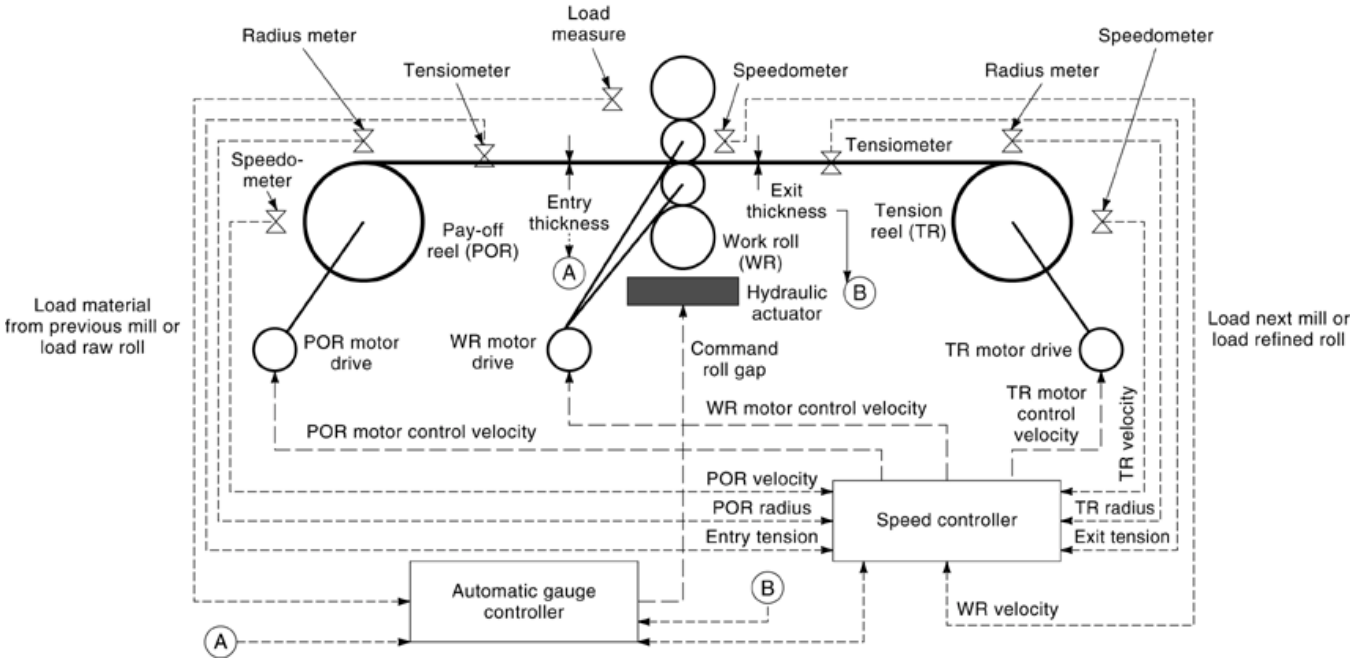

Fig. 2. Steel rolling and pressing mill environment.

Basic Design Style: Multilevel Multistep Design

The attractive basic design style facilitated by *RT* object structuring is to produce a network of *RT* objects meeting the application requirements in a top-down multistep fashion. This will be explained through a simple case study. The *RT* object structuring scheme used in this case study is the *TMO* scheme.

The *RT DCS* to be designed here is a simple manufacturing control system for use in a steel factory. The control system is called an *automatic gauge control* (*AGC*) system and it controls a steel *rolling and pressing process* (RPP). The steel factory application environment considered is shown in Fig. 2. A roll of steel sheet of nonuniform thickness is first loaded onto a cylinder called the *pay-off reel* (*POR*). This *POR* is rotated by an attached *POR* motor drive as shown in Fig. 2. When the *POR* is rotated in a clockwise manner by the attached motor drive, the steel sheet wrapped around the reel advances forward and to the right along a guiding rail (not shown in the figure) and goes between two solid cylinders, known as the *work roll* (*WR*). The *WR* cylinders can not only be rotated by an attached *WR* motor drive but also be pressed together by a hydraulic actuator. Such pressure will be transferred to the steel sheet that passes between the *WR* cylinders. The net effect is to make the thickness of the steel sheet more uniform than before. When the *WR* motor drive rotates the *WR* cylinders, this causes the steel sheet that comes between the cylinders to advance further to the right (away from the WR). Finally, the steel sheet follows the guiding rails (not shown in the figure) and wraps around a cylinder called the *tension reel* (*TR*). As in the case of the *POR* and the WR, the *TR* is also rotated by an attached motor drive, thereby causing the steel sheet to coil around it.

Figure 2 shows just one RPP in detail. In general, the steel mill can have n different RPPs, all arranged in a series and working concurrently. In this case, a roll of raw steel is first loaded on the *POR* of the first RPP. After being processed in the first RPP, the steel is loaded on the *POR* of the next RPP, and so on. The steel sheet thus moves from one RPP to the next in a pipeline fashion until it gets processed by the nth RPP and comes out refined from the pipeline.

Steel Pressing Factory
Access capability (to other TMOs): None
Object data store: 0 to n rolling and pressing processes
SpM (driven by an infinite-precision clock): Update the states of rolling and pressing processes
SvM: Load material

Fig. 3. Steel Pressing Factory *TMO*.

In each RPP, the speeds of all three motor drives as well as the load applied by the hydraulic actuator need to be carefully controlled. For instance, if the *WR* does not have a velocity sufficient enough to keep the sheet between the *POR* and the *WR* taut always, then the material between *POR* and *WR* could bend, resulting in an undesirable situation. On the other hand, if the speeds of the three motor drives are too much off the balance, the steel sheet may be torn apart. The speed controller (*SCT*) shown in Fig. 2, which is to be designed by a computer engineer (team), should control the three motor speeds while the automatic gauge controller (*AGCT*) to be designed should control the load applied to the hydraulic actuator. The three speedometers shown in the figure supply the *SCT* with the current speeds of the motor drives, the two radius meters supply the *SCT* with both the current radius of the *POR* and the radius of the TR, and the two tensiometers supply the *SCT* with the steel tension just before and after the WR. The *SCT* can use these sensor readings to calculate the new speeds of the three motors. The *AGCT* receives the current load applied to the hydraulic actuator and the entry thickness measure of the steel sheet as the input and can use these sensor readings to calculate the new load to be applied to the hydraulic actuator.

Initially the high-level requirement is given by the customer who places an order for the *AGC* system: A rectangular sheet of raw steel of nonuniform thickness, length l m (meter), width w m, and material physical attribute set S, should be refined into a rectangular sheet of steel with at least l m in length, at least w m in width, and thickness in the range of t mm $\pm$ k μm . The minimum thickness of the raw steel is greater than or equal to t mm $-$ k μm.

Step 1: High-Level Specification of the Application Environment of the AGC as a TMO and its Real-Time Simulation. Initially, computer-based controllers do not exist and neither do sensors such as speedometers and tensiometers, and actuators such as hydraulic actuators and motor drives because the system engineer (team) has not decided which types to use. As the first step, the system engineer may describe the application environment of the *AGC* system (i.e., steel pressing factory) as a *TMO* as depicted in Fig. 3. This *TMO* is called the steel pressing factory *TMO*. As we mentioned before, the steel factory consists of up to n RPPs. Hence, the *ODS* of the steel pressing factory *TMO* consists of the state descriptors for (0 to n) RPPs. The information kept in all these state descriptors constitutes the information kept in the steel pressing factory *TMO*.

The RPP state descriptors are periodically updated by a spontaneous method (SpM), that is, a *TT* method. Conceptually, this SpM in the steel pressing factory *TMO* is *activated continuously* and each of its executions is *completed instantly*. The SpM thus represents the continuous state changes that occur naturally in the real RPPs. The service method (SvM) in the steel pressing factory *TMO* functions as an interface to "external clients." The only conceivable client here is the mechanism that inputs the material (i.e., raw steel sheet of nonuniform thickness) to (the *POR* of) the first RPP.

So far, the steel pressing factory *TMO* in Fig. 3 was interpreted as a mere description of the application environment. However, if the activation frequency of each SpM is chosen such that it can be supported by an object execution engine, then the resulting *TMO* becomes a simulation model. The behavior of the application environment is represented by this simulation model somewhat less accurately than by the aforementioned

Rolling and Pressing Process
Access capability (to other TMOs): NextRPP
Object data store: • Pressing mill including POR, WR, TR, sensors, and actuators • Controller (=automatic gauge control + motor speed control)
SpM "Update the descriptors in ODS": • Update the state of pressing mill • Update the state of controller
SvM: Load material

Fig. 4. Rolling and pressing process (RPP) *TMO*.

description model based on continuous activation of SpMs. In general, the accuracy of a *TMO* structured simulation is a function of the chosen activation frequencies of SpMs. Note that this style of simulation is *real-time simulation* in which the simulation objects are designed to show the same timing behavior that the simulation targets do.

Therefore, the *TMO* structuring is effective not only in the multiple-level abstraction of *RT* (computer) control systems under design but also in the accurate representation and simulation of the application environments. This means considerable benefits to the system engineers.

Step 2: High-Level Design of an RPP as a TMO. After creating the high-level specification of the application environment, the system engineer (team) now decides to produce a high-level design of each RPP. For this, the engineer first decomposes the steel factory *TMO* into multiple RPP *TMO*s. Such a decomposition would also involve the introduction of one or more SvMs in each RPP *TMO* to establish some connections among the TMOs and between the TMOs and the clients outside the factory.

Next, the engineer decides on the types of sensors and actuators to be used. Once those devices are chosen, then the control algorithms for operating the devices and controlling the RPP will be determined. Figure 2 already showed all the sensors and actuators chosen. The RPP augmented with chosen sensors and actuators and an imaginery controller can be described as a *TMO* shown in Fig. 4. The *ODS* of this *TMO* contains state descriptors for the pressing mill and the controller that performs the automatic gauge control and the motor speed control. The sensors and actuators are treated as a part of the pressing mill instead of separating them out since their functionality is simple. Therefore, the requirements to be imposed on the computer-based controller have been specified in a concrete form, that is, RPP *TMO* in Fig. 4.

The "update the state of pressing mill" SpM keeps track of the rotary motions of the POR, WR, and TR, and the linear motion of the steel sheet through the guiding rails between the *POR* and the TR. It also keeps track of the action of the hydraulic actuator aimed for setting the roll gap between the *WR* cylinders to the target value ordered by the controller. The "update the state of controller" SpM keeps track of the action of the controller and thus it can be viewed as a core part of the requirement specification for the computer-based controller at this state of the controller development.

Again, the RPP *TMO* contains an SvM representing the operation controlled by an external client of loading material on to the POR. Also, if an RPP has the mechanism for sending processed steel rolls to another RPP, then the *TMO* representing the former RPP should contain the capability for making service calls (i.e., calls for "load material" SvM) to the *TMO* representing the latter RPP. This is the reason for including "NextRPP" in the access capability section of the RPP *TMO* in Fig. 4.

Step 3: Decomposition of the RPP TMO into a Pressing Mill TMO and a Computer-Based Controller TMO. As the system engineer decomposes the single *TMO* representation of the RPP *TMO* in Fig. 4 as a part of more detailed design, a component of the *ODS* becomes a new *TMO*. When these new TMOs are created, the SvMs that serve as the front-end interfaces of these new TMOs should also be created. After the decomposition, the RPP may be composed of a network of two TMOs: the pressing mill *TMO* and the controller *TMO*, which represents the desired actions of both *SCT* and *AGCT*.

In this process, the requirement specifications associated with the controller may be refined. The pressing mill *TMO* may now describe or simulate the actions of the sensors and actuators, the rotary motions of the POR, WR, and the TR, and the linear motion of the steel sheet between the *POR* and the *TR* more accurately than the RPP *TMO* did. Similarly, the controller *TMO* can describe or simulate the desired gauge control and motor speed control actions more accurately than the RPP *TMO* did. Now, the pressing mill *TMO* can be viewed as a description or a real-time simulator of the application environment and the controller *TMO* as an abstract design or requirement specification of the computer-based controller to be implemented by the computer engineer (team).

The full specification of the two *TMO* networks (preferably in the style illustrated later in Fig. 6) plus the following statement can form such a requirements specification: The distance between the highest point on the steel sheet and the lowest point on the steel sheet located either between the *POR* and the *WR* or between the *WR* and *TR* should not exceed h mm.

This requirement is imposed to prevent the undesirable situation of the steel sheet bending in between the *POR* and the *WR* or in between the *WR* and the TR.

Step 4: Further Decomposition of the Pressing Mill TMO and Detailed Design of the Computer-Based Controller. Further decomposition of the pressing mill *TMO* may produce a network of three different TMOs: the *POR TMO*, the *WR TMO*, and the *TR TMO* as shown in Fig. 5. Now, these three TMOs can be hosted on three different nodes if a high-frequency simulation is desired. Here, the *WR TMO* describes or simulates the rotary motion of the WR, the pressing action of the cylinders, the linear motion of steel through the gap between the *WR* cylinders, and the operations of the sensors and actuators located in the vicinity of the WR. The *POR TMO* and the *TR TMO* describe or simulate their corresponding facilities in similar fashions. Each of these environment TMOs should have SvMs that interface not only with the computer-based controller but also with other environment *TMO*s.

During this step the computer engineer (team) may produce a more detailed design specification of the computer-based controller by expanding the single *TMO* design into a network of two TMOs as shown in Fig. 5: the *automatic gauge controller* (*AGCT*) *TMO* and the *speed controller* (*SCT*) *TMO*. These two TMOs may be hosted on two different computer nodes or on the same computer node. The detailed design specification of the *SCT TMO* is shown in Fig. 6.

The *SCT TMO* receives from the *POR TMO* the sensor readings, namely, the current *POR* velocity, the current *POR* radius, and the current steel tension in the measurement point between the *POR* and the WR. It also receives the current *WR* velocity from the *WR TMO*. In addition, it receives from the *TR TMO* some sensor readings such as the current *TR* velocity, the current *TR* radius, and the current steel tension in the measurement point between the *WR* and the TR. The *SCT* then calculates the velocity required for the *POR* motor drive, the velocity for the *WR* motor drive, and the velocity for the *TR* motor drive that satisfy the requirement specification and outputs these values to appropriate motor drives.

The *AGCT TMO* receives from the *WR TMO* some sensor readings such as the current value of load applied to the *WR* cylinders and the thickness of the steel sheet measured in the entry point of the WR. This *TMO* then calculates the desired load to be applied to the cylinders.

The *AGC* system in Fig. 5 thus consists of five interconnected *TMO*s. The upper portion depicts the three TMOs that compose the *RT* simulator of the application environment. The lower portion depicts the two

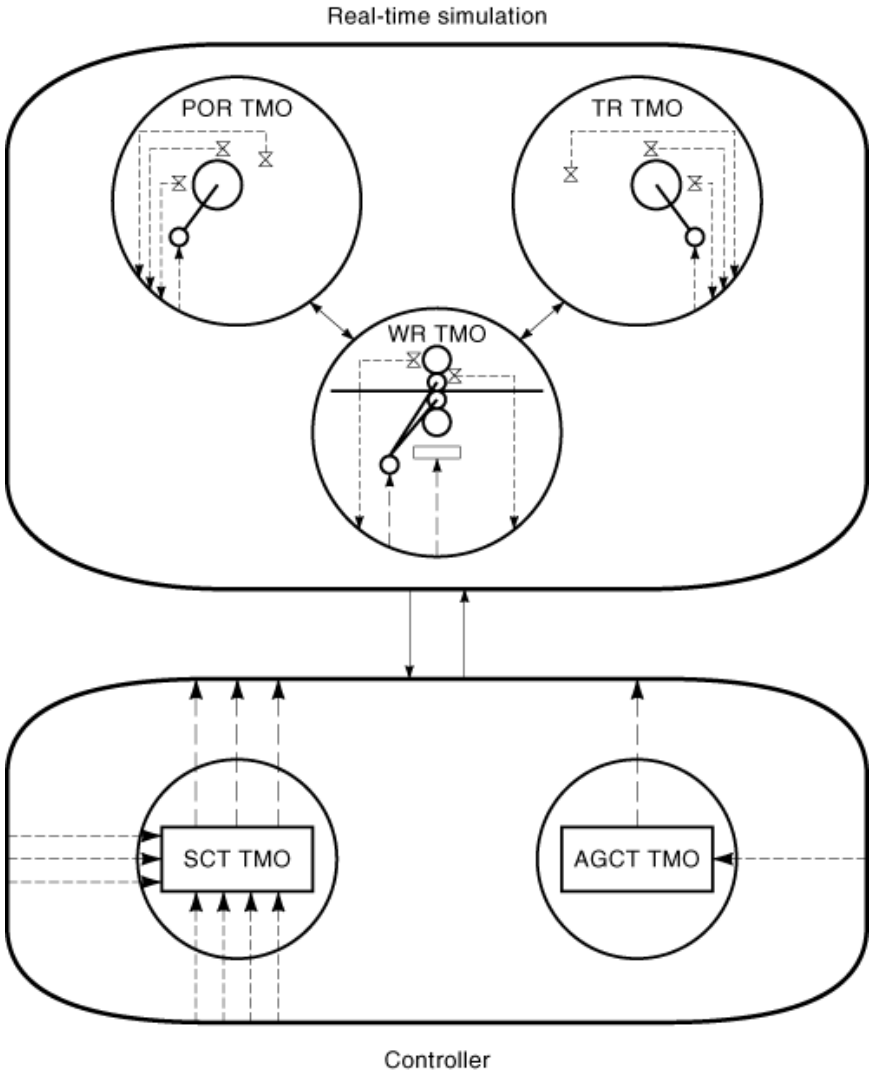

Fig. 5. Network of TMOs: The *AGC* system.

TMOs that compose the computer-based controller. The *AGC* system implemented can be easily expanded with functions such as handling of various alarm conditions including a safe shutdown of the pressing mill.

Advantages of the RT-Object-Based Multilevel Multistep Design. Under the multilevel multistep design approach illustrated above, both the computer engineer (team) who produces control computer systems and the system engineer (team) who interfaces with the customers of the computer-embedded application systems and produces precise specifications of requirements to be met by the computer engineer, use the same structuring approach during their systematic construction of specifications.

The computer engineer receives a RT-object-structured requirement specification. The computer engineer initially produces an abstract single RT-object design and then proceeds to refine it into a more detailed design which has the structure of an RT-object network (e.g., Fig. 5).

Speed Controller

Access capability (to other TMOs):
- Pay-off reel (POR) (Receive_motor_drive_command_velocity_from_Speed_Controller_TMO)
- Work roll (WR) (Receive_motor_drive_command_velocity_from_Speed_Controller_TMO)
- Tension reel (TR) (Receive_motor_drive_command_velocity_from_Speed_Controller_TMO)

Object data store:

```
SCT_Data_Table:list of{POR_radius, POR_velocity, Entry_tension; WR_velocity;
    TR_radius, TR_velocity, Exit_tension;}
```

SpM:

SpM 1: Send command_velocities_to_motor_drives
- Read SCT_Data_Table from ODS.
- Calculate the POR motor drive command velocity and send it to POR motor drive (via SvM request).
- Calculate the WR motor drive command velocity and send it to WR motor drive (via SvM request).
- Calculate the TR command velocity and send it to TR motor drive (via SvM request).

TT: for T = from TMO_START + WARMUP_DELAY_SECS to TMO_START + SYSTEM_LIFE_HOURS
every PERIOD
start-during (EST, LST) finish-by T + deadline

Output: (to POR TMO)<T, T + xxx msec> POR motor drive command velocity
(to WR TMO)<T, T + yyy msec> WR motor drive command velocity
(to TR TMO) <T, T + zzz msec> TR motor drive command velocity

SvM:

SvM 1: <Accept-via-Service_Request_Channel-with-Delay_Bound-of ACCEPTANCE_DEADLINE
<start-within INITIATION_DEADLINE under MAX_REQUEST_RATE finish-within
EXECUTION_TIME_LIMIT>
Receive_Sensor_Outputs_from_Pay_Off_Reel_TMO
- Receive from POR TMO POR_velocity, POR_radius, and Entry_tension.
- Update SCT_Data_Table in ODS.

Concurrency: Other SvM 1 invocations are not in place.
Input: POR radius, velocity, and entry tension
Output: None

SvM 2: <Accept-via-... >
Receive_Sensor_Output_from_Work_Roll_TMO
...
SvM 3: <Accept-via-... >
Receive_Sensor_Output_from_Tension_Reel_TMO
...

Fig. 6. Speed controller *TMO*.

On the other hand, the system engineer first starts with a single RT-object representation of the application environment (the RPP) (e.g., Fig. 3) in which sensors, actuators, and control computer systems are to be embedded. The system engineer gradually refines this single RT-object representation into an RT-object network representation (e.g., Fig. 4), which can be given to the computer engineer as a requirement specification.

Thereafter, the system engineer (or another team) can optionally refine the environment portion of the RT-object network representation into a *RT* simulator of the application environment, for example, simulator depicted in (the upper half of) Fig. 5. The environment simulator can then be used for testing the control computer system produced. The environment simulator is a *real-time simulator* that produces sensor data in real time, takes real-time commands for actuators, and simulates the subsequent operations of actuators in real time. Obviously, this kind of testing yields better coverage than the testing based on non-real-time simulation of the application environment.

BIBLIOGRAPHY

1. A. Attoui An object oriented model for parallel and reactive systems, *Proc. IEEE CS 12th Real-Time Syst. Symp.*, 1991, pp. 84–93.
2. T. Bihari P. Gopinath K. Schwan Object-oriented design of real-time software, *Proc. IEEE CS 10th Real-Time Syst. Symp.*, 1989, pp. 194–201.
3. G. Booch *Object Oriented Analysis and Design with Applications*, 2nd ed., Redwood City, CA: Benjamin/Cummings, 1994.
4. O. J. Dahl Hierarchical Program Structuring, in Dahl, Dijkstra, and Hoare (eds.), *Structured Programming*, New York: Academic, 1972.
5. Y. Ishikawa H. Tokuda C. W. Mercer An object-oriented real-time programming language, *IEEE Comput.*, **25** (10): 66–73, 1992.
6. K. H. Kim et al., Distinguishing features and potential roles of the RTO.k object model, *Proc. 1994 IEEE CS Workshop Object-Oriented Real-Time Dependable Systems (WORDS)*, Dana Point, 1994, pp. 36–45.
7. K. H. Kim Object structures for real-time systems and simulators, *IEEE Comput.*, **30** (8): 62-70, 1997.
8. S. Kleiman et al, *Programming with Threads*, Mountain View, CA: Sunsoft Press, 1996.
9. H. Kopetz K. H. Kim Temporal uncertainties in interactions among real-time objects, *Proc. IEEE CS 9th Symp. Reliable Distributed Systems*, AL, 1990, pp. 165–174.
10. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, 1995.
11. J. Richter *Advanced Windows*, 3rd ed., Redmond, WA: Microsoft Press, 1997.
12. B. Selic G. Gullekson P. T. Ward *Real-Time Object-Oriented Modeling*, New York: Wiley, 1994.
13. R. Soley (ed.), *Object Management Architecture Guide*, 3rd ed., New York: Wiley, 1995.
14. K. Takashio M. Tokoro DROL: An object-oriented programming language for distributed real-time systems, *Proc. ACM OOPSLA*, 1992, pp. 276–294.

K. H. KIM
University of California