

Fault-Tolerant Real-Time Objects

The authors attack a challenging and important problem.

K.H. (Kane) Kim and Chittur Subbaraman

BOTH THE COMPLEXITY OF LARGE-SCALE REAL-time (RT) computer systems in safety-critical application fields and the reliability expectations of the user community for such computer systems have been growing fast in recent years. The reliability goals that customers have started imposing on such systems cannot be adequately met by relying on the conventional design technologies, which tend to be weakly scaleable and do not have enough in common with the general object-oriented (OO) design technology increasingly applied to the production of large-scale non-RT business and office data processing software. In order to achieve significant improvement in design efficiency and system reliability attained in the real-time computing application fields, we believe that it will be most rewarding and timely to establish the following types of technologies:

- *General-form design style:* Real-time computing must be realized in the form of a generalization of non-real-time computing, as opposed to the form looking like an esoteric specialization [6].
- *Design-time guarantee of timely service capabilities of subsystems:* To meet the demands of the general public on the assured reliability of future real-time computing systems (RTCSs) in safety-critical applications, the system engineer must produce design-time guarantees for timely service capabilities of various subsystems (which will take the form of objects in object-oriented system designs) as opposed to relying on the testing only.
- *Scalable time-bounded fault tolerance scheme:* Ideally, fault detection and recovery actions must always be executed such that intended output actions of real-time computations take place on time. Such an idealistic type of fault

tolerance, which is to accomplish all critical actions (i.e., output actions of critical real-time tasks) successfully in spite of component failures, is called the *action-level fault tolerance* (ALFT) [5]. The timing properties of a scheme for achieving ALFT should be easily analyzable and in particular, such a scheme must yield a small recovery time-bound for non-negligible types of fault scenarios. Desirable fault tolerance techniques must be scalable in that they must be applicable to various distributed and/or parallel computer systems of different sizes.

There may be many approaches to realizing each of these three goals. However, there are few concrete demonstrated approaches. Moreover, what we ultimately need is an integrated design technology that meets all three goals mentioned previously. Establishing such a technology is among the most challenging open research issues in the area of reliable real-time computing. In this article, we state some major issues and establish some feasible directions to search for such a technology.

To facilitate the general-form design style, a powerful computation structuring scheme uniformly applicable to both real-time applications and non-RT applications must be established. Such a technology will stimulate the formation of a synergistic relationship between the RTCS development community and the non-RT business data processing system development community. This article briefly reviews some recently established approaches for extending the conventional object structuring scheme into a powerful scheme for structuring both real-time and non-RT computations. These powerful structuring schemes may be called real-time object structuring schemes. Of these schemes, those that enable the system designer to produce design-time guarantee of timely service capabilities of

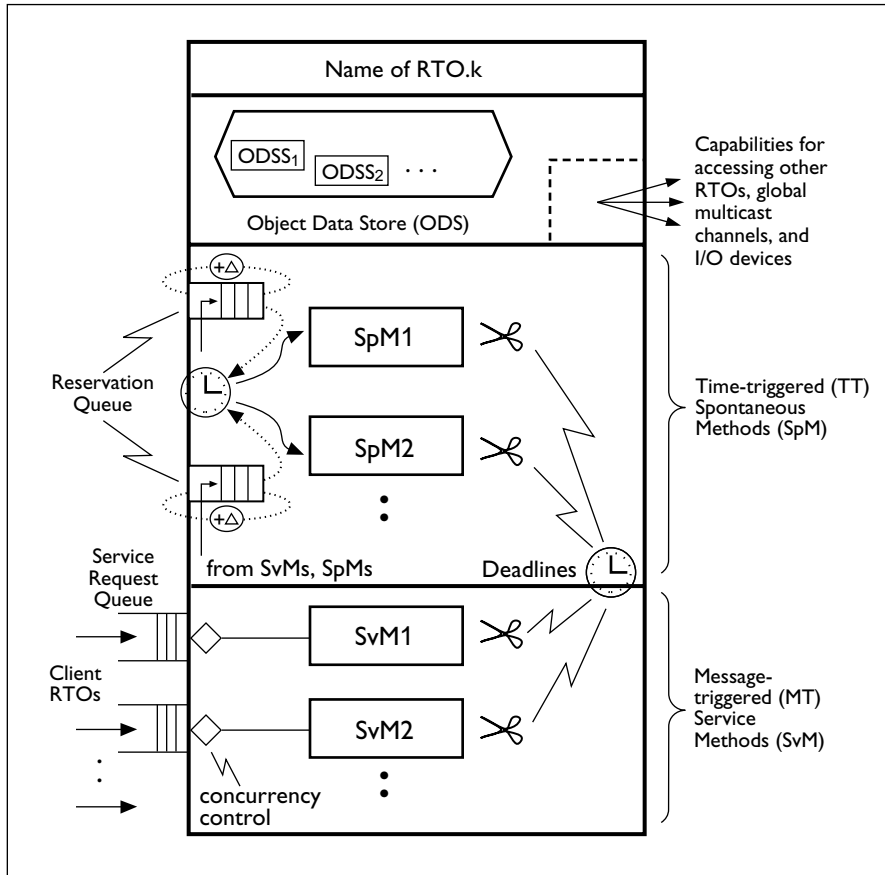


Figure 1. The RTO.k object structure

objects with moderate efforts are considered the most desirable. This is because experiences of practicing engineers indicate that testing alone is not sufficient for assuring the level of reliability of RTCSs demanded by customers.

Moreover, real-time fault tolerance capabilities must be built into RTCSs in safety-critical applications in order to realize an acceptable level of reliability. As mentioned earlier, the fault-tolerant system design must enable rigorous validation of the acceptable recovery time-bound for non-negligible types of fault scenarios. Realizing such time-bounded fault tolerance capabilities in RTCSs of distributed and/or parallel computing type has been a great challenge and realizing it in object-based RTCSs has been an even greater challenge. However, two recent technological changes have converted the challenge into a short-term or near-term synthesis problem:

- (1) Recent maturing of the time-bounded fault tolerance scheme applicable to process-structured RTCSs [5].
- (2) Recent emergence of real-time object structuring schemes, especially those enabling the efficient production of timely service guarantees associated with the objects at the design time.

We present a basic scheme for realizing real-time object replicas which possess time-bounded fault tolerance capabilities. This object-based, real-time fault tolerance scheme is called the *primary-shadow RTO.k replication* (PSRR) scheme and has been validated through several real-time computing application development experiments conducted in the authors' laboratory. A few industry research organizations have recently adopted the PSRR scheme. The PSRR scheme validated in the laboratory is an integration of the primary-shadow active replication approach established in development of the process-structured real-time fault tolerance schemes [5] and the specific real-time object structuring scheme called the *RTO.k object scheme* [6, 7]. However, the basic principles of the PSRR scheme are mostly applicable to other

real-time object structuring schemes as well. For a clear understanding of the basic principles of the PSRR scheme, the RTO.k object-based active replication scheme will be used to illustrate the principles throughout this article.

The core of the PSRR scheme consists of two sets of rules. First, there are basic operational rules under which the primary real-time object and the shadow object should cooperate. Secondly, there are basic rules for structuring real-time object methods to compose primary and shadow partner real-time objects. Before we present these rules, we review the essence of the RTO.k object structuring scheme.

Essence of the RTO.k Object Structuring Scheme

In the last several years, there has been a growing trend of research activities for extending the conventional OO-structuring approaches to support RTCS design [1, 4, 7] Among the object extensions developed, the RTO.k object model, also called the *time-triggered real-time object* (TT-RTO) model, is unique in the extent of enabling the design-time guarantee of timely service capabilities of objects. Models and prototype implementations of the effective operating system (OS) support and the friendly application programming interface (API) have been devel-

oped [8, 9]. The basic structuring rules and the execution semantics associated with the RTO.k object are reviewed in this section.

The General-form Design Style Facilitated

The basic structure of an RTO.k object is depicted in Figure 1. Significant extensions of the conventional object model that are incorporated into the RTO.k object model are the following:

(a) Spontaneous methods (SpMs) clearly separated from service methods (SvMs): for some methods of an RTO.k object, a real-time clock serves as the mechanism for triggering the method executions as the clock reaches some values specified at design time. Such methods are called time-triggered (TT-) methods, also called the SpMs, and clearly separated from the conventional SvMs triggered by messages from clients. The two method types in an RTO.k object 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 SpMs.”

Triggering times for SpMs must be fully specified as constants at design time. Those real-time constants appear in the first clause of an SpM specification called the autonomous activation condition (AAC) section. For example, an AAC may be “for $t = \text{from } 10\text{am to } 10:50\text{am every } 30\text{min start-during } (t, t+5\text{min}) \text{ finish-by } t+10\text{min}”$. 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 (or another SpM) within the same RTO.k object requests future executions of a specific SpM.

(b) For each execution of a method of an RTO.k object, a deadline is imposed.

Design-time Guarantee of Timely Service Capabilities

The designer of each RTO.k object provides a guarantee of timely service capabilities of the object by indicating the deadline for every output produced by each SvM (and each SpM that may be executed on request from SvMs) in the specification of the SvM (and some relevant SpMs) advertised to the designers of potential client objects. An execution rule has been incorporated into the RTO.k object model in order to reduce the designer’s efforts in determining these deadlines. It is called the *basic concurrency constraint* (BCC) and prevents data conflicts between SpMs and SvMs. 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. Therefore, SpMs are

given higher priorities for execution over the potentially conflicting SvMs. This priority and the fact that triggering times of SpMs are fixed at the design time make it very easy to analyze the execution time behavior of SpMs.

Primary-Shadow Active Replication Principle

The *primary-shadow active replication principle* is the basic underlying principle of a process-structured real-time fault tolerance scheme named the DRB/PSP scheme, which has been developed over the past 13 years [5]. A natural way to incorporate the active replication principle into an object structuring scheme is to replicate each object to form a pair of partner objects and execute the partners in two different nodes. The methods of the primary object alone will produce all external outputs under normal circumstances. However, since each partner object has its own object data store (ODS), the methods of both objects perform the ODS updates.

System Model Considered and Fault Types Covered

The system that we consider consists of a general network of RTO.k objects which interact with one another. An RTO.k network of objects may be hosted on many different hardware platforms. A typical hardware node may consist of one or more processors, all accessing one or more global shared memory modules through a common bus. Various nodes may be interconnected with each other through a local area network (LAN) or a wide area network. Each processor in a node may host multiple SpMs or SvMs of the object. The ODS segments (ODSS) of the object may be mapped to the shared memory modules of the node. The communication between methods of two different RTO.k objects hosted on the same node can be achieved through message channels called *data field channels* (DFCs) laid over the shared memory whereas the communication between methods of two different RTO.k objects hosted on two different nodes can be achieved through DFCs laid over the interconnection network [9].

The PSRR scheme has been developed to tolerate the following types of faults:

(a) Acceptance test (AT) failure including deadline violations: The AT of a method execution is an execution of a routine or an OS primitive designed to check the reasonableness of the computation results of the method execution. It consists of both the *logic AT* and the *timing AT*. Therefore, the AT can result in a logic AT failure indicating the incorrectness of the logical value of the result produced, which may be due to transient hardware faults or design faults in the method computation procedure. Or it can result in a timing AT failure indicating a violation of

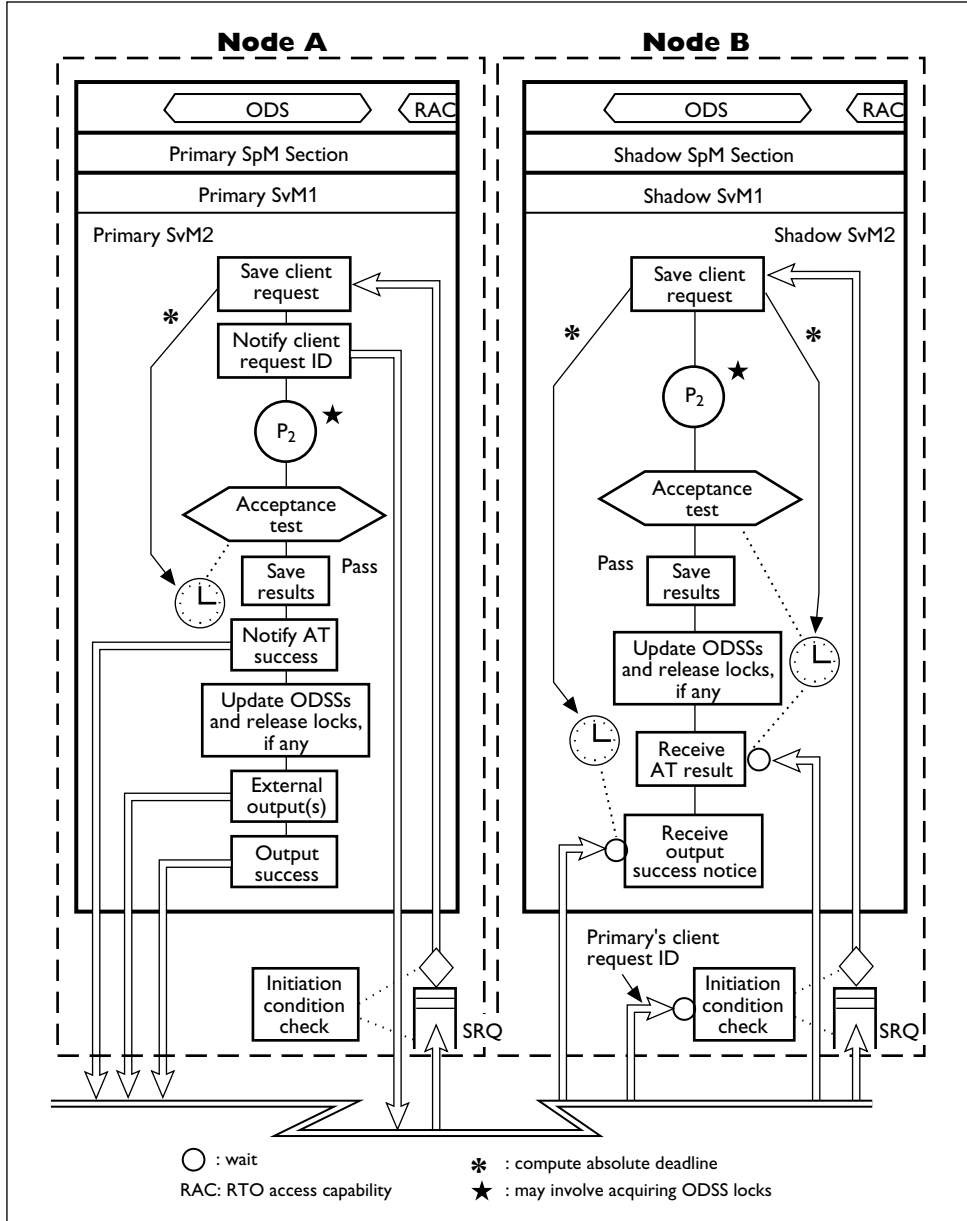


Figure 2. A fault-free method execution cycle of SvMs in a primary and shadow RTO.k object

the method execution deadline;

(b) Crash of a method execution caused by faults in hardware, OS, etc.;

(c) ODSS loss primarily due to the (crash) faults in the shared memory modules hosting ODSSs: If an ODSS loss is detected, then the entire node is treated as a crashed node under the PSRR scheme;

(d) Node crash;

(e) Loss of messages caused by faults in communication links and nodes. Message acknowledgment schemes deal with these faults but for highly effective handling of these

faults and node crashes, a suspicious method execution manager needs to seek the advice of a *network surveillance and reconfiguration* (NSR) manager. An NSR manager can facilitate fast learning of faults or repair completion events by each interested fault-free node and fast reconfiguration—we presume the presence of an NSR manager. Both centralized and decentralized techniques for NSR have been explored in the past [3, 5, 10].

Basic Operational Rules of the PSRR Scheme

An RTO.k object method in general consists of computations interleaved with input actions and output actions. Here the input action may be an internal input action or an external input action. An internal input action is a reading of the content of an ODSS after locking the ODSS. An external input action may be:

- (a) receiving a service request from a client,
- (b) receiving a reply message from an SvM to which this method has requested for service, or

(c) receiving data from the application environment via a sensor.

Similarly, the output action may be an internal output action or an external output action. An internal output action is the release of locks on ODSSs after updating the ODSSs. An external output action may be:

- (a) a signal to the application environment,
- (b) a service call to another RTO.k object, or
- (c) returning a reply to a client.

For simplicity of discussion, we consider the types of methods in which all output actions are performed at the end. Other types of methods are considered later. Figure 2

shows a fault-free execution cycle of an SvM in a primary RTO.k object and the corresponding execution in the shadow partner RTO.k object where the two partner RTO.k objects are hosted on two different nodes connected to a local-area network. Both the primary SvM execution (to be exact, the execution of the SvM in the primary RTO.k) and the shadow SvM execution use the same version (P2) of the computation procedure.

Node A hosts the initial primary RTO.k object and node B hosts the initial shadow RTO.k object. Both nodes receive the same client request from the network in their appropriate service request queues (SRQs). After the client request arrives, the initiation condition check (ICC) mechanism of the execution engine in the primary node periodically checks whether the required primary SvM execution can be initiated without violating the BCC. As soon as the ICC initiates the primary SvM execution, the client request message is saved into some log cache, the storage of which can survive at least as long as the node hosting the RTO.k object does. The reason for saving this message will become evident later.

The next step in primary SvM execution is to inform the shadow RTO.k of the ID of the client request that the former has chosen to process. When the ICC in the shadow node receives the client request ID from the primary SvM execution, it starts periodically checking whether the shadow SvM execution can be initiated without violating the BCC. The shadow SvM execution initiated by the ICC starts by saving the client request message into its log cache and then processes the client request corresponding to the received ID.

The primary and shadow SvM executions process the client request concurrently and invoke their self-checking independently and concurrently by using the same AT routine. This routine can be implemented entirely in software. Or some errors may be detected by the mechanisms in the execution engine (hardware plus OS) before execution of the AT routine but such error detection can be treated in the same manner as the failure of the AT. Since the primary and the shadow SvM executions both pass the AT in Figure 2, they independently save their computation results into their log cache. Upon saving the results, the primary SvM execution informs the shadow partner of its success of the AT. The former then performs all of its output actions (which may include external output actions as well as internal output actions such as the release of ODSS locks after updates). The primary SvM execution may involve seeking the help of an NSR manager to confirm the successful execution of its external output actions. Once the primary SvM execution confirms the successful delivery of its output, it sends an output success notice to the shadow SvM execution. The shadow SvM execution, after saving all of its computation results into the log cache, proceeds to perform its

internal output action (i.e., updating the variables of any pertinent ODSSs and releasing the locks on the ODSSs), but upon receiving the AT success notice from the primary partner, it skips the external output step and waits for the primary partner's output success notice.

Let us now consider a different method execution scenario in which the primary SvM execution fails in the AT but the shadow SvM execution passes. As soon as the primary SvM execution fails its AT, it should inform its shadow partner of the failure and change its role to that of the shadow. As a new shadow in recovery mode, it should then make a retry of the processing of the client request after rolling back to the beginning and retrieving the saved client request message from the log cache. This is why saving the client request message at the beginning is necessary in both the primary and shadow SvM executions. In response to the primary's AT failure notice, the shadow SvM execution will change its role to that of the primary and perform the appropriate external output actions, which can be viewed as a forward recovery action for the PS-RTO.k object pair. As a part of the role change, the old primary SvM execution should set the object mode flag in its owner RTO.k object to indicate the new role assumed. This in effect makes "the entire RTO.k object change its role". The old shadow SvM execution should do the same.

Apart from an explicit AT failure notice from the primary SvM execution, the shadow SvM execution may learn about the failure of the primary SvM execution in one of the following ways:

- (a) absence of a client request ID from the primary within a specific deadline which results in the ICC in the shadow node triggering the initiation of the local SvM as the primary SvM execution,
- (b) absence of an AT result notice from the primary within a specific deadline, or
- (c) absence of an output success notice from the primary within a specific deadline.

In Figure 2, both the primary and shadow SvM executions use the same computation procedure. This makes the PS object pair intolerant to any fault in the computation procedure design since the same design fault will be present in both the primary and shadow SvMs [2,11,12]. To ameliorate this problem, we may embed software redundancy into each SvM by using the *recovery block structuring scheme* [12]. A recovery block may consist of multiple versions of a method procedure and an acceptance test (AT) procedure. Such versions are called try blocks. In most cases a recovery block containing just two try blocks, a primary try block and an alternate try block, is designed. Under the PSRR scheme, the roles of the two try blocks are assigned differently in the two objects (and hosting nodes). The governing rule is that the primary method tries to execute the primary try block whenever possible, whereas the shadow

In order to establish some pictorial representations, let us first consider methods of type-a. Figure 3(a) depicts the structure of a primary method and its shadow partner by using the notations. As shown, the primary method first saves the initial input (the client request message in the case of an SvM and the initiation timestamp in the case of an SpM) into the log cache. Then it acquires the locks for two ODSSs, $ODSS_1$ and $ODSS_2$. It then proceeds to execute its computation procedure M1. Later it takes an external input action and again saves the input data into the log cache. After passing the AT, it saves the results into the log cache and notifies the shadow partner of the AT result. It then performs the output actions successfully and sends an output success notice to its partner. The structural representation of the shadow method execution represented in Figure 3(a) is also self-evident.

The reason for saving the external input data into the log cache is that if the method-segment which involves an input action fails the AT during the first try, then the received input message is necessary during the retry procedure.

If a method execution fails after saving the method computation results into the log cache but before performing all of its internal output actions (e.g., the primary method execution updating $ODSS_2$ and releasing its lock in Figure 3(a)), then the engine supporting the method execution should detect the failure and restart the method execution from the point at which the result saving was completed. Upon restarting, the method execution determines whether an ODSS has remained locked instead of blindly reissuing the corresponding ODSS update action. If the ODSS has remained locked, then the ODSS update can be reissued and the execution can proceed. Otherwise, it is clear that the ODSS update and release action had been completed before the failure of the method execution. Therefore, the retry execution must proceed without reissuing the ODSS update. If the method execution fails after saving the method computation results into the log cache but before performing all of its external output actions, an NSR manager should detect the failure and inform the engine supporting the method execution of this. In response, the execution engine should restart the method execution as a new shadow from the point at which the result saving was completed. Since the role of the method execution is that of the shadow during the retry procedure, no external output actions are reissued.

Structuring of Type-B Methods

Figure 3(b) depicts the structure of a method of type-b. This method does not contain any branch statement. The computation procedure used by this method involves an output action at an intermediate point. Figure 3(b) illustrates a rule under which, before any intermediate output action is taken, the reasonableness of the output data pre-

pared is determined by execution of an intermediate AT, AT_k, and the complete computation state of the method is saved. The complete computation state saved includes the states of all the variables, including the local variables of the method. This state saving is in good contrast to the saving of the computation results which follows the final AT. When the final AT has succeeded, there is no need to save the states of all the variables that have been updated during the method execution since the local variables of the method will no longer be used. Only the computation results need to be saved into the log cache.

If the method execution fails after the state-saving point, then it can roll back to the state saved and retry the computation. Therefore, an intermediate output action is never revoked. Upon completing an intermediate AT, the primary method execution sends the AT result to the shadow partner.

In a sense, every intermediate output action serves as a boundary between two method-segments (i.e., one method-segment ending immediately before the output action and the follow-on method-segment beginning immediately after the output action). Each intermediate AT may involve referencing the values of the variables saved at the most recent state-saving point as well as the input data saved into the log cache. For software fault tolerance, at least two try blocks for each method-segment should be designed.

If an object method contains a large number of intermediate output actions, each at a different execution point, then the burden of designing intermediate ATs and alternate try blocks becomes very large. Therefore, there are incentives to the PS-RTO.k replica designer for either grouping several intermediate output actions into one large intermediate output action or converting intermediate output actions to final output actions to the extent possible.

The presence of branch statements adds a new dimension in structuring (intermediate and final) ATs and try blocks for method-segments. New issues arise where some branches involve some intermediate output actions as depicted in Figure 3(c). The method in the figure involves two branches, one branch (B1) involving an intermediate output action and another branch (B2) involving no intermediate output action. Under the structuring rule explained earlier, an intermediate AT must be inserted before the intermediate output action occurring in B1. This intermediate AT should be designed to check if the method-segment M1 ranging from the method initiation point to the point at which the intermediate AT can begin, has been executed properly and has produced an acceptable result. Included here is the important confirmation that the correct branch, B1, has been taken.

When structuring the final AT, the AT must be designed to check at the minimum if some reasonableness conditions which must be met regardless of which branch

has been taken, have indeed been met. In addition, the AT can be designed to check for different additional conditions when different branches are taken.

Structuring of Type-c Methods

This is the most general case. Figure 3(d) illustrates one such primary method. Here, the ODSSs are acquired and released in the middle of the method execution. In addition, the method also makes service requests to other RTO.k objects (shown as IO1) and receives return-results (shown as IO2) in the middle of the execution. Although this case appears to have a more complicated structure than the cases in Figures 3(a), 3(b), and 3(c), it does not necessitate any additional structuring rule.

Experimentation with the PSRR Scheme

In an effort to validate the PSRR scheme, several experimental developments of fault-tolerant real-time computing application systems have been made in the authors' laboratory. The information on the software tools used to support these developments is available from the Web page of the authors (www.eng.uci.edu/dream). Similar experimental developments are under way in several other organizations (e.g., SoHaR, Inc. in Los Angeles). One of the implemented fault-tolerant RTO.k structured applications is a C3 defense application running on PCs interconnected by an Ethernet LAN. The RTO.k object structuring scheme has been used to structure not only the control computer system, but also the real-time simulators of the environmental objects such as radar, airplanes, and so forth. A selected RTO.k object has been made tolerant to hardware faults by using the PSRR scheme. Transient as well as permanent hardware faults were injected and the response of the primary-shadow object pair was observed. The object pair was able to recover from the injected faults in time to produce all the outputs within the specified deadlines.

Conclusion

Object-oriented design has become a mainstream technology for developing large-scale software for non-RT business and office data processing applications. On the other hand, it is fair to say that impacts of the object-oriented design on development of RTCs has been relatively minor so far. Major changes here are expected soon since practical tools for robust structuring and reliable execution of real-time object-based distributed software systems are becoming available. Robust structuring of fault-tolerant real-time object-based systems is not a mature technology area yet, but time is ripe for intensive research for maturing it. In this article, we presented the primary-shadow active replication principle and basic approaches for its exploitation in real-time object based fault-tolerant system structuring. We believe these basic principles provide a sound basis for estab-

lishing a robust technology for designing real-time object-based distributed computing systems that possess time-bounded fault tolerance capabilities. The basic principles presented here may be largely applicable to the environments where conventional (passive) object structuring approaches are used but validating this conjecture appears to be a meaningful topic for future research. □

REFERENCES

1. Attoui, A. An object oriented model for parallel and reactive systems. In *Proceedings of IEEE CS 12th Real-Time Systems Symposium* (1991), pp. 84–93.
2. Bastani, F. et al. Toward dependable safety-critical software. In *Proceedings of IEEE CS Workshop on Object-oriented Real-Time Dependable Systems (WORDS '96)* (Laguna Beach, Calif., Feb. 1–2, 1996), pp. 86–92.
3. Hecht, M. et al. A distributed fault tolerant architecture for nuclear reactor and other critical process control applications. In *Proceedings of IEEE CS 21st International Symposium on Fault-Tolerant Computing (FTCS-21)* (Montreal, June 1991), 1991, pp. 462–469.
4. Ishikawa, Y., Tokuda, H., and Mercer, C. W. An object-oriented real-time programming language. *IEEE Computer* (Oct. 1992), 66–73.
5. Kim, K.H. Action-level fault tolerance. In *Advances in Real-Time Systems*, S.H. Son, Ed., Prentice Hall, 1994, 415–434.
6. Kim, K.H. et al. Distinguishing features and potential roles of the RTO.k object model. In *Proceedings of IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS '94)* (Dana Point, CA, Oct. 24–25, 1994), pp. 36–45.
7. Kim, K.H. and Kopetz, H. A Real-time object model RTO.k and an experimental investigation of its potentials. In *Proceedings of IEEE CS Computer Software and Applications Conference (COMPSAC)* (Taipei, Taiwan, Nov. 1994), pp. 392–402.
8. Kim, K.H. et al. A timeliness-guaranteed kernel model—DREAM kernel and implementation techniques. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications (RTCSA)* (Tokyo, Japan, Oct. 1995), pp. 80–87.
9. Kim, K.H. et al. The DREAM library support for PCD and RTO.k programming in C++. In *Proceedings WORDS '96*, (Laguna Beach, CA, Feb. 1–2 1996), pp. 59–68.
10. Kopetz, H. and Grunsteidl, G. TTP-A: Time triggered protocol for fault tolerant real-time systems. In *Proceedings of IEEE CS 23rd Fault Tolerance Computing Symposium* (Toulouse, France, June 1993), pp. 524–533.
11. Lyu, M.R., Ed. *Software Fault Tolerance*. John Wiley and Sons, England, 1995.
12. Randell, B., and Xu, J. The evolution of the recovery block concept. In *Software Fault Tolerance*, Michael R. Lyu, Ed., 1995, 1–21.

K.H. (KANE) KIM (kane@ece.uci.edu) is a professor in both the Department of Electrical and Computer Engineering and the Department of Information and Computer Science at the University of California, Irvine. **CHITTUR SUBBARAMAN** (csubbara@ece.uci.edu) is a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of California, Irvine.

The research work reported here was supported in part by U.S. Navy, NSWC Dahlgren Division under Contract No. N60921-92-C-0204, in part by the University of California MICRO Program under Grant No. 96-169, in part by the California Transportation Department via the UCI Institute for Transportation Studies, in part by Hitachi, Ltd, in part by ETRI, and in part by LG Electronics.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/97/0100 \$3.50