Maximizing Concurrency and Analyzable Timing Behavior in Component-Oriented Real-Time Distributed Computing Application Systems

K. H. (Kane) Kim and Juan A. Colmenares
University of California, Irvine, USA
{khkim, jcolmena}@uci.edu

Demands have been growing in safety-critical application fields for producing networked real-time embedded computing (NREC) systems together with acceptable assurances of tight service time bounds (STBs). Here a service time can be defined as the amount of time that the NREC system could take in accepting a request, executing an appropriate service method, and returning a valid result. Enabling systematic composition of large-scale NREC systems with STB certifications has been recognized as a highly desirable goal by the research community for many years. An appealing approach for pursuing such a goal is to establish a hard-real-time (HRT) component model that contains its own STB as an integral part. The TMO (*Time-Triggered Message-Triggered Object*) programming scheme is one HRT distributed computing (DC) component model established by the first co-author and his collaborators over the past 15 years. The TMO programming scheme has been intended to be an advanced high-level RT DC programming scheme that enables development of NREC systems and validation of tight STBs of such systems with efforts far smaller than those required when any existing lower-level RT DC programming scheme is used. An additional goal is to enable maximum exploitation of concurrency without damaging any major structuring and execution approaches adopted for meeting the first two goals. A number of previously untried program structuring approaches and execution rules were adopted from the early development stage of the TMO scheme. This paper presents new concrete justifications for those approaches and rules, and also discusses new extensions of the TMO scheme intended to enable further exploitation of concurrency in NREC system design and programming.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.1.5 [**Programming Techniques**]: Object-oriented programming; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms: Software components, hard real-time, service time bound

Additional Key Words and Phrases: basic concurrency constraint (BCC), deadline, guaranteed execution duration bound (GEDB), message-triggered, non-blocking buffer, ordered isolation (OI) rule, service methods, spontaneous methods, start-time window, time-triggered, TMO

---

1. INTRODUCTION

In recent years *networked real-time embedded computing* (NREC) applications started showing explosive growth in both variety and production volume. The growth is evident even in safety-critical applications. As a consequence, demands for improving both productivity of the NREC software engineers and reliability of the produced NREC application systems have started increasing rapidly.

One fundamental way toward meeting these demands is to increase the level of abstraction dealt with in programming of NREC systems to one much higher than that in the currently widely practiced programming that uses functions, threads, task priorities, and sockets as basic building-blocks. The goal of upgrading the level of programming primitives and languages is to substantially reduce the amount of labor required in NREC programming while drastically increasing the readability and analyzability of the produced programs. To meet the productivity improvement demands properly, a high-level real-time (RT) distributed computing (DC) programming approach should facilitate systematic composition of NREC applications from independently validated components. The key step in developing such a programming approach is to establish a high-level RT DC component model.

Complementary advances must also occur in another area to properly meet the product reliability demands. That is, reliable NREC systems must be accompanied by certificates of their reliability. In particular, reliable NREC systems must be produced with acceptable assurances of tight *service time bounds* (STBs). Each STB is a tight upper bound on the amount of time that the NREC system could take in accepting a request (that has just arrived at the incoming door of the host node, e.g., incoming message queue), executing an appropriate service method, and returning a valid result (to the outgoing door of the hosting node, e.g., network interface unit or outgoing message queue). A *service time* can also be defined more broadly as the duration from the instant at which a certain event of interest occurs to the instant at which a valid response is output. Enabling systematic composition of large-scale NREC systems with STB certifications has been recognized as a highly desirable goal by the research community for at least 20 years.

A conceptually natural and appealing approach for pursuing such a goal is to establish a component model containing its own STB as an integral part. Such a component model may be called the *hard-real-time (HRT) component model.* The component model should have a structure that enables easy stacking or other more complex forms of interconnecting many different instances of the model. An STB of an NREC system composed of HRT components can then be established relatively easily by a control-flow-path analysis and summation of the STBs of selected components. Starting with such STB as a guide, a tighter STB can be established via various means, e.g., an hybrid approach that combines analysis and measurements [Im and Kim 2006].

The composition process is recursive in nature. Level-N HRT components can be established first and then by interconnecting those, Level-(N-1) HRT subsystems can be built. By using Level-(N-1) HRT subsystems as components and interconnecting them in some ways, Level-(N-2) HRT subsystems can be built. Repeating this way, the full system, which can be viewed as a Level-0 HRT subsystem, can be built.

The first co-author established the initial skeleton of an HRT component model, called TMO (*Time-Triggered Message-Triggered Object*) [Kim 1997; 1999; 2000; Kim et al. 2005], 15 years ago and since then the model and supporting tools have been enhanced at several steps by him and his collaborators. TMO is a syntactically simple and natural but semantically major extension of the conventional object structure. The TMO programming and specification scheme is also considered one of the most advanced attempts for raising the level of abstraction at which RT DC programmers are allowed to exercise their logic. The TMO scheme is meant to relieve the NREC application designers and programmers of the burden of dealing with low-level programming tools and low-level abstractions of computing and communication environments. TMO programmers are required to specify both the interactions among DC program components, i.e., TMOs, and the timing requirements of various actions in natural intuitively appealing forms only.

In devising and enhancing the TMO scheme, the most important goals, after that of raising the level of abstraction in RT DC programming, have been the following:

(1) To enable relatively easy analysis of timing behavior, in particular, derivation of tight STBs, of TMOs and total NREC application systems structured as TMO networks; and

(2) To enable extensive exploitation of concurrency within NREC application systems.

Various mechanisms and various rules for executing parts of TMOs were incorporated into the TMO scheme with the above two goals in mind. The purpose of this paper is to present the mechanisms newly introduced in recent years for further satisfying the two goals mentioned above. Strengthened justifications for those mechanisms and rules which were presented before are also provided in this paper.

In Section 2, an overview of the basic TMO programming scheme is given and then the TMO scheme is compared against another well-established HRT component-based programming scheme. A number of previously untried program structuring approaches and execution rules which were adopted from the early stage of development of the TMO scheme are discussed in Section 3. The main goal in this section is to present some new concrete justifications for those approaches. In Section 4, some new extensions of the TMO model which are intended to enable further exploitation of concurrency in NREC system design and programming are presented. Section 5 is a follow-on to Section 3 and provides a new concrete substantiation of one basic concurrency control rule adopted in the TMO model. Section 6 provides a conclusion.

2. OVERVIEW OF THE BASIC TMO PROGRAMMING SCHEME

The TMO scheme is a general-style RT DC extension of the pervasive object-oriented and component-based design/programming approach [Kim 1997; 2000; 2002a]. It has been established to facilitate RT DC software engineering in a form which software engineers experienced in the vast non-RT software field can adapt to with small effort. Calling the TMO scheme a high-level DC programming scheme is justified by the following characteristics of the scheme:

(1) *No manipulation of processes and threads.* Concurrency is specified in an abstract form at the level of object methods. Since processes and threads are transparent to TMO programmers, the priorities assigned to them, if any, are not visible, either.

(2) *No manipulation of hardware-dependent features in programming interactions among objects.* TMO programmers are not burdened with any direct use of low-level network protocols and any direct manipulation of physical channels and physical node addresses.

(3) *Timing requirements need to be specified only in the most natural form of a time-window for every time-triggered method execution and a completion deadline for every client-requested method execution.* This high-level expression matches the most closely with the designer's intuitive understanding of the application's timing requirements.

Once the high-level specification of timeliness requirements is registered with the middleware supporting TMOs, then the middleware does its best to meet the specification by using the CPU scheduler and other resource schedulers in the underlying node OS kernel and network infrastructure. Here optimal ways of scheduling both computing and communication resources have been insufficiently studied and are open to continuous research [Kim and Liu 2002; Kim 2004; Shin and Lee 2004; Li 2005].

The TMO scheme was meant to be a HRT component model from its birth. The TMO scheme and the TTP programming scheme (developed by Kopetz and his collaborators [Kopetz and Grünsteidl 1994; Kopetz et al. 1997; Eberle et al. 2001]) are two of the very few practical RT component-based programming schemes that have been formulated from the beginning with the objective of enabling *design-time guaranteeing of timely actions.* The TMO scheme incorporates several rules for execution of parts of TMOs and the intention is to make the design and validation of TMOs together with STBs relatively easy and systematic while not reducing the programming power in any significant way [Kim 1997; 1999; 2002a].

2.1 TMO Structure and Design Paradigms

As depicted in Figure 1, the basic TMO structure consists of four parts:

(1) **ODS-sec** (*Object-data-store section*). This section contains the data-container variables shared among methods of a TMO. Variables are grouped into *ODS segments* (ODSSs) which are the units that can be locked for exclusive use by a TMO method in execution. Access rights of TMO methods for ODSSs are explicitly specified and the execution engine (a composition of networked hardware, node OS, and middleware) analyzes them to exploit maximal concurrency.

(2) **EAC-sec** (*Environment access capability section*). Contained here are "*gate objects*" providing efficient call-paths to remote TMO methods, logical multicast channels called RMMCs (*Real-time Multicast and Memory Replication Channels*) [Kim 2000; Kim et al. 2005], and I/O device interfaces.

(3) **SpM-sec** (*Spontaneous method section*). It contains *time-triggered methods* whose executions are initiated during specified time-windows. Time-triggered

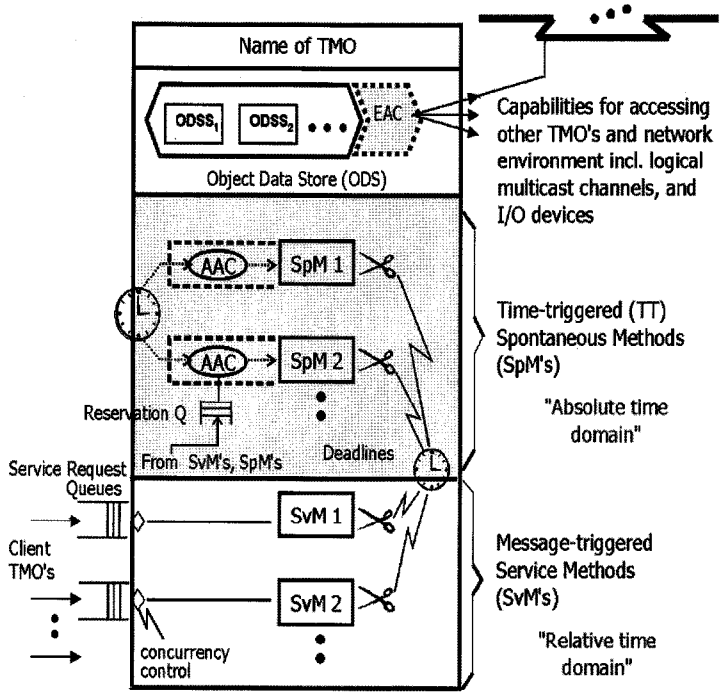

Figure 1. Basic TMO structure (adapted from [Kim 1997]).

(or spontaneous) methods will be discussed further later.

(4) **SvM-sec** (*Service method section*). It contains service methods which can be called by other TMOs.

The key features of the TMO programming scheme are reviewed here.

Use of a Global Time Base. All time references in a TMO are references to *global time* [Kopetz 1997] in that their meaning and correctness are unaffected by the location of the TMO. If GPS receivers are incorporated into the TMO execution engine, then a global time base of microsecond-level precision can easily be established. Within a cluster computer or a LAN-based DC system a master-slave scheme, which involves time announcements by the master and exploitation of the knowledge on the message delay between the master and the slave, can be used to establish a global time base of sub-millisecond level precision [Kim et al. 2002].

RT DC Component. TMO is a DC component and thus TMOs distributed over multiple nodes may interact via *remote method calls.* Non-blocking types of remote method calls are supported to allow concurrent execution of client methods in one node and server TMO methods in different nodes or the same node.

TMOs can use another interaction mode in which messages can be exchanged over logical multicast channels of which access gates are explicitly specified as data members of involved TMOs. The channel facility is called the *Real-time Multicast*

and Memory-replication Channel (RMMC) [Kim 2000; Kim et al. 2005], of which an earlier version was called the HU data field channel [Kim et al. 1995]. The RMMC scheme facilitates RT publisher-subscriber channels in a versatile form. It supports not only conventional *event messages* but also *state messages* based on distributed replicated memory semantics [Kopetz 1997].

Natural-form Specification of Timing Requirements and Guarantees. TMO has been devised to contain only high-level intuitive and yet precise expressions of timing requirements. *Start-time-windows* and *completion deadlines* for object methods are used but no specification in indirect terms (e.g., priority) are required. A completion deadline may be specified in the form of a global time instant (e.g., 10:30am) or a bound on execution time (e.g., 250 milliseconds) spent after a signal triggering the method execution activation arrives at the host node. The latter bound is called the *guaranteed execution duration bound* (GEDB). GEDBs associated with a TMO are taken as guaranteed STBs by the designers of the clients of the TMO. *Deadlines for result arrivals* can also be specified in the client's calls for service methods.

Autonomous Active DC Component. The autonomous-action capability of the TMO stems from one of its unique parts, called the *time-triggered (TT) methods* or *spontaneous methods* (SpMs), which are clearly separated from the conventional *service methods* (SvMs). The SpM executions are triggered upon reaching of the global time at specific values determined at design time whereas the SvM executions are triggered by service request messages from clients. For example, the triggering times of an SpM may be specified as:

```
FOR t = FROM 10:00am TO 10:50am
  EVERY 30min
  START-DURING (t,t+5min)
  FINISH-BY (t+10min)
```

This specification of the execution-time window of an SpM is called the *Autonomous Activation Condition* (AAC) of the SpM and has the same effect as the following does:

```
{[START-DURING (10:00am,10:05am) FINISH-BY 10:10am],
 [START-DURING (10:30am,10:35am) FINISH-BY 10:40am]}
```

By using SpMs, *global-time-based coordination of distributed computing actions* (TCoDA), a principle pioneered by [Kopetz and Ochsenreiter 1987; Kopetz 1997], can be easily designed and realized.

Basic Concurrency Constraint (BCC). It is a major execution rule intended to enable reduction of the designer's efforts in guaranteeing timely service capabilities of TMOs and it prevents potential 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.* The full set of data members in a TMO is called an *object data store* (ODS). An ODS is declared as a list of *ODS segments* (ODSSs), each of which is thus a subset of the data members in the ODS and is accessed by concurrently running object-method

executions in either the *concurrently-reading* mode or the *exclusive-writing* mode. Thus, an SvM is allowed to execute only if no SpM that accesses the same ODSSs to be accessed by this SvM has an execution time-window that will overlap with the execution time-window of this SvM. The BCC does not reduce the programming power of TMO in any way (Section 3 presents a concrete substantiation for this claim).

Power of TMO Network Structuring. An underlying design philosophy of the TMO scheme is that an RT computing application system will always take the form of a network of TMOs, which may be produced in a top-down multi-step fashion [Kim 1997]. All conceivable practical RT and non-RT applications can be built as TMO networks.

2.2 TMO Execution Engine and Programming Tools

TMO programming has been enabled without creation of a new language or compiler. Instead, a middleware model called the TMOSM (*TMO Support Middleware*) provides execution support mechanisms and can be easily adapted to a variety of commercial kernel-hardware platforms in wide use in industry [Kim et al. 1999; Kim 2002a; Jenks et al. 2007]. TMOSM uses well-established services of commercial OS kernels, e.g., process and thread support services, short-term scheduling services, and low-level communication protocols, in a manner transparent to the application programmer. Prototype implementations of TMOSM currently exist for Windows XP, Windows CE, and Linux 2.6 platforms.[1] Along with TMOSM, the *TMO Support Library* (TMOSL) has been developed [Kim 1999; 2000; Kim et al. 2005]. It provides a set of friendly application programming interfaces (APIs) that wrap the execution support services of TMOSM. TMOSL defines a number of C++ classes and enables convenient high-level programming by approximating a programming language directly supporting TMO as a basic building block. Other research teams have also developed TMO execution engines based on different kernel platforms [Kim et al. 2002; Kim et al. 2005].

Since its first introduction, the TMO programming model has been enhanced in several steps along with TMOSM and TMOSL. The TMO programming scheme and supporting tools have been used in a broad range of basic research and application prototyping projects in a number of research organizations. New-generation application demos developed in the last few years include cars that can be driven by drivers located thousands of miles away, a dancing robot squad, wireless-network based digital music ensemble, high-QoS multimedia streaming synchronization, and tiled display capable of playing high-definition movies. The programming tools have also been used in an undergraduate course on RT DC programming at UCI for some years.

2.3 Comparison of Two Hard Real-time Component-based Programming Schemes: TTP and TMO

The TTP (*Time-Triggered Protocol*) programming scheme developed by Kopetz and his collaborators [Kopetz and Grünsteidl 1994; Kopetz et al. 1997; Eberle

[1] Available at: `http://dream.eng.uci.edu/TMOdownload/`

et al. 2001] is another well-established HRT component-based NREC programming approach. The main differences between the two programming schemes are two-fold. First, the TTP programming scheme is meant to be a relatively low-level programming scheme. In the TTP scheme, a function running on an RT thread is the basic programming unit and functions running on different nodes interact via one-way messages. In the case of the TMO scheme, the basic component is the TMO which is an object structure encompassing RT service methods and TT methods and TMOs can interact via remote method calls and RMMCs.

Secondly, in the TTP scheme, all task scheduling is done off-line, i.e., static scheduling is used. Even individual message communications over a shared bus are scheduled at design time. In the TMO scheme, some high-level scheduling such as periodic activations of TT methods is done at design time but activations of service methods, executions of both segments of TT methods and segments of service methods, and message communications are scheduled at run time.

With the TTP scheme, application designers can produce STBs with higher precision than when using the TMO scheme. On the other hand, being a higher-level programming scheme, the TMO scheme can lead to smaller amount of labor required in producing many complex NREC applications. Which approach requires greater efforts for design and validation of STBs during the development of various applications is a topic for future research.

3. CONCURRENCY RULES IN THE TMO MODEL FOR EASING THE DERIVATION OF HARD BOUNDS FOR SERVICE TIMES

3.1 Types of Concurrency and Unlimited Concurrency in Absence of ODSS Conflicts

In an NREC system structured as a TMO network, concurrency may be exhibited among TMOs. From the vewpoint of a TMO, all methods in other TMOs are treated as remote methods regardless of whether the latter TMOs are located within the same DC node or in different DC nodes. SpMs belonging to different TMOs may be active concurrently. Also, SvMs belonging to different TMOs may be active concurrently. They could have been called by remote SpMs, or some of them could have been activated by remote SvMs via non-blocking calls.

Even within a TMO, the following major types of concurrency may be found:

(C1) *Concurrency among SpM executions*, specified in an implicit but natural manner. For example, one SpM designed to be triggered at 10:00am and another SpM at 10:01am, each with a GEDB (*Guaranteed Execution Duration Bound*) of 5 minutes.

(C2) *Concurrency among SvM executions.*

(C3) *Concurrency between SpM executions and SvM executions.*

The TMO scheme adopted the approach of allowing unlimited concurrency as long as there is no data conflict among the candidates for concurrent executions. This is the opposite of an approach directly indicating that unit A and unit B may proceed in parallel. Therefore, each method in a TMO is registered with the execution engine and the registration packet includes the IDs of the ODSSs to be accessed by the method being registered. The intended mode of using each ODSS,

i.e., *read-only* or *read-write*, is also registered. The execution engine can then figure out whether or not to activate a certain method at a given time by checking if any of the ODSSs needed by the candidate method is currently held by any on-going method execution.

The TMO scheme allows *pipelined execution of SvMs.* That is, when a call for an SvM comes in while the SvM is already in execution due to an earlier call, an execution of the SvM corresponding to the later call may be initiated and the two executions of the same SvM may proceed in a pipelined fashion as long as there is no data conflict. If the SvM uses ODSSs in the read-only mode, then pipelined execution will be allowed since there cannot be any data conflict. However, if the SvM uses an ODSS in the read-write mode, then there is no possibility for pipelined execution.

While facilitating maximal exploitation of concurrency is a good thing, determining tight STBs of server TMOs involves analyses of the competitions among various TMOs and parts of TMOs for using execution engine facilities. Multiple TMOs can be co-resident in a processing node of a distributed and/or parallel processing execution engine. Even when only one TMO is resident on an execution engine node, methods of the TMO compete among themselves for obtaining execution engine node services.

3.2 Basic Concurrency Constraint (BCC)

The design and validation of a GEDB of an SpM become greatly simplified by incorporating the BCC execution rule. Although type-C3 concurrency, i.e., concurrency between SpM executions and SvM executions, is allowed, SpM executions cannot be directly disturbed by any SvM execution under BCC. If there is an SpM that accesses the same ODSSs to be accessed by the subject SvM and current activation of an execution of the SvM will create the possibility of the SvM execution being overlapped with an execution of the SpM, the activation of the SvM execution will be delayed. Therefore, in validating a GEDB of an SpM, the efforts can be focused on possible competitions between the subject SpM and other SpMs as well as the effects of allocating and multiplexing some execution engine resources among various method executions which do not have data conflicts.

BCC allows analysis of SpM execution times without being concerned with the SvM execution times and also allows analysis of SvM execution times based on the knowledge of SpM execution times. It thus enables step-by-step analysis and contributes to major reduction of the burdens imposed on the designer in determining GEDBs and STBs.

Although the proposition that *BCC does not reduce the programming power of TMO in any way* has been accepted as a valid fact because counter examples have not been found, a clean substantiation has not been presented until this time. A concrete argument substantiating the proposition is provided in Section 5.

3.3 Ordered Isolation (OI) Rule

The difficulty of analyzing method-to-method competitions depends much on the way ODSSs are locked and released. To reduce that difficulty further after incorporating BCC, the TMO scheme adopted the *ordered isolation* (OI) rule [Kim 2002b]. The OI rule can be stated by using the term *initiation timestamp* (*I-timestamp*)

defined as follows:

- In the case of an SvM execution, the *I-timestamp* is defined as the record of the time instant at which the execution engine initiated the SvM execution after receiving the client request and ensuring that the SvM execution can be initiated without violating BCC.
- In the case of an SpM execution, the *I-timestamp* is defined as the record of the time instant at which the SpM execution was initiated according to the AAC specification of the SpM.

The OI rule has the following two parts:

(OI-1) A method execution with an older *I-timestamp* must not be waiting for the release of an ODSS held by a method execution with a younger *I-timestamp*.

(OI-2) A method execution must not be rolled back due to an ODSS conflict.

One can easily see that if these rules are not followed, then the validation of GEDBs can be drastically more complicated. The price paid for reducing the complexity of deriving tight *execution duration bounds* (EDBs) by adopting the OI rule is the loss of some concurrency.

The OI rule also dictates that each used ODSS is locked and released exactly once during a method execution.

A conservative procedure for the execution engine to follow for locking ODSSs as a part of activating an execution of a TMO method while satisfying the OI rule is presented below. This procedure is *conservative* in that under it a method execution is activated only after all the ODSSs to be used by the method are locked for use in the method execution. A practical advantage of this approach is that the execution engine does not have to deal with the situation where a method execution starts and then gets blocked later because an ODSS needed in the next instruction step is still held by another method execution with an older *I-timestamp*. Derivation of a tight EDB is also easier than when a less conservative procedure is used.

`Procedure OL`: *Locking ODSSs under OI*

The subject method to be activated is an SvM, called `SvM1`. `ODSS-set(method1)` denotes the set of ODSSs to be used during an execution of `method1`. `ITS(method2)` denotes the *I-timestamp* of an execution of `method2` that is either on-going or to be activated. `GEDB(method3)` is the guaranteed execution duration bound of `method3`. Finally, `SxM` denotes a TMO method (i.e., an SpM or SvM) in general, and so does `SzM`.

`Begin-Procedure`

(1) Do the BCC check. That is, see if there is any SpM whose start-time-window falls partly or wholly within the next `GEDB(SvM1)` amount of time from now. If there is such an SpM, keep the `SvM1` execution on the list `To-Activate` until the SpM execution is finished. Then go back to restart Step 1. If there is no such SpM, create the *I-timestamp* `ITS(SvM1)` and continue.

(2) For each `SxM` execution that has not been activated, i.e., has not fully locked `ODSS-set(SxM)` yet, and has `ITS(SxM)` older than `ITS(SvM1)`, do the following:

(a) Check if `ODSS-set(SvM1)` overlaps with `ODSS-set(SxM)`.
   i. If there is an overlap, the execution engine keeps the `SvM1` execution in the list `To-Activate` until the `SxM` execution is activated and finished. Thereafter, continue.
   ii. If there is no overlap, continue.

(3) Try to lock `ODSS-set(SvM1)` by doing the following steps and when it is successful, the `SvM1` execution is activated.
   (a) Try to put a preliminary lock on each member of `ODSS-set(SvM1)` by processing the set in the order of the ODSS IDs.
      i. If preliminary locks have been put on all members of `ODSS-set(SvM1)` without encountering any problem, convert all preliminary locks into real locks. The activation of the `SvM1` execution is now successful.
      ii. Suppose that a problem is encountered, i.e., a certain member of the set, say `ODSS9`, is still held by an older on-going method execution, say `SzM`, and at least one of the two method executions, `SvM1` and `SzM`, uses `ODSS9` in the *read-write* mode. In this situation, keep the `SvM1` execution on the list `To-Activate` until the `SzM` execution is finished. When the `SzM` execution is finished and thus `ODSS9` is released, check for a possible BCC conflict with a younger SpM execution. Under normal circumstances, i.e., if `GEDB(SvM1)` was specified properly, such an SpM should not be found at this point since Step 1 was already executed. In the unusual case where such an SpM is found, report the anomaly to the *fault-management system* if there is one, and go back to restart Step 1. In a normal case, go back to execute the remainder of Step 3a (i.e., to lock remaining members of `ODSS-set(SvM1)`).

`End-Procedure`

If the subject method is an SpM, the procedure becomes somewhat simpler but it will not be described here due to space limit.

With the conservative `Procedure OL` in use, each method execution keeps its ODSS set locked for its entire duration. The benefit is again the relative ease in deriving a tight EDB of a TMO method. Search for approaches that are less restrictive than the OI rule and `Procedure OL` and yet do not make the derivation of tight EDBs of a TMO much harder is a subject for future research.

3.4 Maximum Invocation Rate (MIR)

The TMO programming scheme requires specification of the *maximum invocation rate* (MIR), i.e., the maximum rate at which the server TMO can receive service requests from client objects, to be associated with each SvM. If service requests from client objects arrive at a server TMO at a rate exceeding the MIR indicated in the specification for the server TMO, then the execution engine of the server TMO may return exception signals to the client objects. This is another rule necessary to make the derivation of hard STBs feasible. The NREC system designer should ensure that the aggregate arrival rate of service requests at each server TMO does not exceed the MIR during any period of system operation.

4. NEW EXTENSIONS OF THE TMO MODEL FOR FURTHER INCREASE OF CONCURRENCY

In this section, new extensions of the TMO model that enable deeper exploitation of concurrency without making derivation of STBs much harder are discussed.

4.1 Early Release of ODSSs

By default, the ODSSs held by a TMO method execution are all released together when the method execution is finished. A way to increase concurrency further is to allow a TMO method to release the ODSSs early, i.e., before the method execution is finished. Therefore, such a mechanism, represented by API `ReleaseODSS()`, has been incorporated into the TMO model.

Such early release of an ODSS can result in activating another method execution requiring the ODSS before the releasing method execution terminates. Therefore, the `Procedure OL` discussed in Section 3.3 needs to be extended accordingly if early release of ODSSs is allowed. Due to space limit, such an extended procedure will not be discussed here. If an SvM uses an ODSS in the *read-write* mode, then pipelined executions of the same SvM are not possible at all, but with early release of such an ODSS, pipelined execution is enabled to some extent.

It is important to note that once a method execution releases an ODSS, it cannot lock the ODSS again due to the OI rule and other reasons. A companion of the ODSS release operation is the operation represented by API `RW2RO()` that changes the access mode for the subject ODSS from the *read-write* mode to the *read-only* mode. This means that the read-write lock that the method execution has held on the ODSS is changed into the read-only lock. This change cannot be reversed during the remainder of the method execution. In a sense, it is a half-way release. Again, by changing the lock this way, it may allow other TMO method executions to start early.

4.2 Underground Non-Blocking Buffer (NBB) with a Pair of Access Gates

In the basic TMO scheme, there is no way for any two concurrent method executions to exchange any data. This is because an ODSS cannot be shared when at least one method execution needs to access it in the *read-write* mode. Even with the ODSS release mechanism discussed in Section 4.1, data can be passed only once and only in one direction from the earlier started producer (method execution) to the later started consumer. There is no way to enable multiple rounds of data passing in both directions between two concurrent method executions.

In some application cases, it may be desirable to let two long-running SpM executions exchange data streams. For example, an SpM running at the periodic execution rate of 30 executions per second and producing a video data stream by operating a camera may want to pass the data stream to another SpM running at the rate of 25 executions per second and rendering the data stream on an LCD monitor.

Allowing a TMO method execution to dynamically lock and release an ODSS more than once conflicts directly with our goal of enabling relatively easy design and validation of STBs of TMOs. When multiple method executions become interdependent through such mechanisms, the derivation of any reasonably tight STB

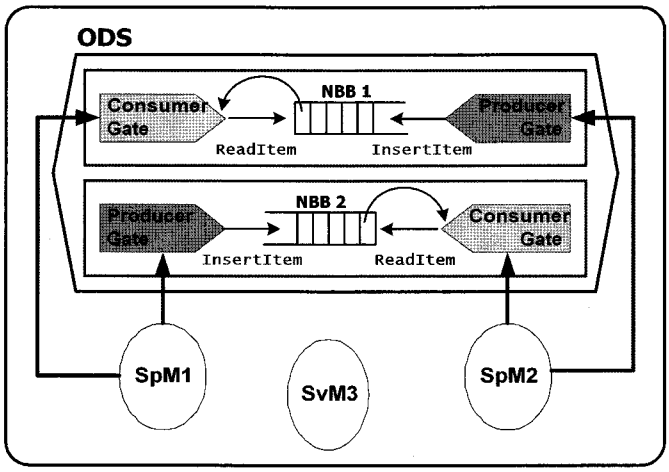

Figure 2. Underground NBBs in a TMO.

of the TMO can often become a hopeless task.

A new mechanism that enables multiple rounds of data passing from one method execution to the other method execution and yet does not damage the nature of the TMO scheme which makes STB validation relatively easy is proposed here. The mechanism is based on the *Non-Blocking Buffer* (NBB) developed in recent years by several teams [Varma 2001; Kim 2006; Kim et al. 2007]. An NBB used between a producer thread and a consumer thread allows the producer to insert a new data item into its internal circular buffer at any time without experiencing any blocking. Instead of using either synchronization constructs (e.g., locks and monitors), which sometimes lead a thread to a blocked state, or special atomic instructions (e.g., `test-and-set` and `compare-and-swap`), NBB relies only on the atomicity of reads and writes of aligned single-word integer variables. If the internal circular buffer is saturated, then the producer attempting to insert a new item can detect it immediately and choose to do other things for a while and then check the NBB again.

Similarly, the NBB allows the consumer to retrieve a data item from the internal circular buffer at any time without experiencing any blocking. If the internal circular buffer is empty, then the consumer attempting to retrieve an item can detect the emptiness immediately and choose to do other things for a while and then check the NBB again. Incidentally, most textbooks covering the subject of operating systems contain discussions on the producer-consumer problem but give the erroneous impressions that it is inevitable to use lock-based synchronization constructs and encounter blockings in solving the problem.

The version of NBB that is appropriate for use between two methods in a TMO is depicted in Figure 2. Two NBBs are there. Each NBB consists of an internal circular buffer, a *producer gate*, and a *consumer gate*. The two gates are ODSSs and they are registered with the execution engine. In a sense, the internal circular buffer is treated as an invisible data structure. Therefore, the producer method puts a read-write lock on the producer gate and the consumer method puts a read-write

lock on the consumer gate and then the two are treated by the execution engine as two independent methods not sharing any data structure. Only the TMO designer knows that the internal circular buffer is a shared data structure but the execution engine *pretends* not to know this and allows the producer method execution and the consumer to proceed concurrently. Therefore, this version of NBB may be called the *underground NBB.*

The producer gate, which is an ODSS and thus an object, provides a method for inserting an item into the circular buffer. When the producer (TMO method) is registered with the execution engine, the registration packet includes the ODSS ID of the producer gate. Also, the consumer gate provides a method for retrieving an item from the circular buffer, and when the consumer (TMO method) is registered with the execution engine, the registration packet includes the ODSS ID of the consumer gate. An appropriate set of template classes in C++ for the underground NBB has been devised, but the classes are not discussed here due to space limit. When an underground NBB is instantiated, the two gates are created.

In principle, an underground NBB may be used between any pair of TMO methods, i.e., (SvM1, SvM2), (SvM1, SpM2), and (SpM1, SpM2). However, its use between two SvMs is likely to be rare and this conjecture needs to be confirmed or disproved in future experimental research.

5. SUBSTANTIATION FOR THE BCC PROPOSITION

A concrete substantiation for the following proposition can now be provided by use of the underground NBB mechanism discussed in Section 4.2.

Proposition: *BCC does not reduce the programming power of TMO in any way.*

Substantiation: The only restriction in concurrency that BCC introduces is the possible delay in activating an SvM execution incurred due to a potential data conflict with an SpM execution. This delay could be avoided in the absence of BCC. If avoiding such delay is important for a certain SvM, then one can easily produce such a TMO without removing BCC. For example, see Figure 3. Both the SpM and the SvM there use the ODSS. If it is desirable to let the SvM start as early as the SpM does and let both the SvM and the SpM proceed concurrently, one might envision removing BCC with concomitant loss of benefits and then designing both TMO methods to dynamically and harmoniously share the data produced by them. Instead, Figure 4 depicts an organization of a TMO which produces such effects without violating BCC or any other rules associated with the TMO scheme. The SpM is the only possible user of: i) `ODSS1`, ii) the producer gate of underground `NBB1`, and iii) the consumer gate of underground `NBB2`; whereas the SvM is the only possible user of: i) `ODSS2`, ii) the consumer gate of underground `NBB1`, and iii) the producer gate of underground `NBB2`. Therefore, the execution engine treats the SpM and the SvM as two independent methods. The SvM can start as early as the SpM does and the two method executions can proceed concurrently. Whenever the SpM produces new data, it puts the data into not only `ODSS1` owned by itself but also `NBB1` from which the data becomes accessible to the SvM. Similarly, whenever the SvM produces new data, it puts the data into not only `ODSS2` owned by itself

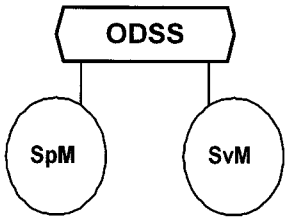

Figure 3. An SpM and an SvM using an ODSS.

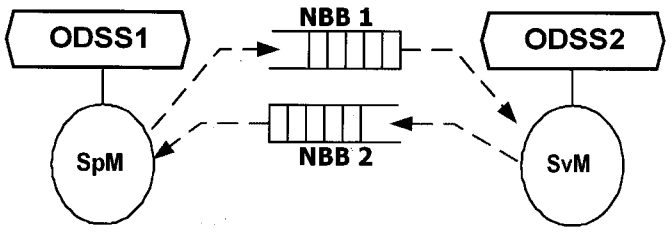

Figure 4. Multiple rounds of cooperation between two on-going method executions in a TMO.

but also NBB2 from which the data becomes accessible to the SpM. Therefore, the two TMO method executions can fully share and exchange data as they proceed concurrently without violating BCC or any other execution rules. Consequently, the proposition is valid.

6. CONCLUSION

The motivations underlying various structuring principles and execution rules in the TMO programming scheme have been discussed in this paper. The TMO scheme has been intended to be a frontier high-level RT DC programming scheme. This has been the premier goal not to be compromised easily. The second important goal has been to enable design and validation of tight STBs of NREC systems with efforts far smaller than those required when using any existing lower-level RT DC programming scheme. Yet, we have not wanted to give up the use of run-time scheduling approaches for allocating computation and communication resources to various RT computation segments, mainly because the gap in achievable resource utilization between pure static scheduling approaches and the approaches involving run-time scheduling actions has been judged to be too much to ignore up to this time.

A challenge has been to enable maximum exploitation of concurrency without damaging any major structuring and execution approaches adopted for meeting the first two goals. With those goals set, a number of previously untried program structuring approaches and execution rules were adopted from the early development stage of the TMO scheme. These approaches and rules were discussed in this paper along with some new concrete justifications. Moreover, new arguments substantiating the fact that the BCC (*Basic Concurrency Constraint*) rule does not reduce the programming power of TMO in any way were presented.

This paper also discussed new extensions of the TMO model intended to enable

further exploitation of concurrency in TMO-structured NREC system design and programming; they are: (1) the mechanisms for early release of ODSSs, and (2) the underground NBB (Non-Blocking Buffer), which is a mechanism that enables multiple rounds of data passing between two TMO method executions. Effective applications of these mechanisms are meaningful topics for future research. Also, the design and validation of tight STBs of NREC systems are still a big and immature technological area inviting massive investment of research efforts.

ACKNOWLEDGMENTS

The research reported here is supported in part by the NSF under Grant Numbers 03-26606 (ITR) and 05-24050 (CNS). Juan A. Colmenares also thanks the University of Zulia (LUZ) for supporting his participation in this research. No part of this paper represents the views and opinions of the sponsors mentioned above.

REFERENCES

EBERLE, S., EBNER, C., ELMENREICH, W., FÄRBER, G., GÖHNER, P., HAIDINGER, W., HOLZMANN, M., HUBER, R., SCHLATTERBECK, R., KOPETZ, H., AND STOTHERT, A. 2001. Specification of the TTP/A protocol. Research Report 61/2001, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria.

IM, C. AND KIM, K. H. 2006. A hybrid approach in TADE for derivation of execution time bounds of program-segments in distributed real-time embedded computing. In *Proceedings of the 9th IEEE Int'l Symposium on Object and Component Oriented Real-Time Distributed Computing (ISORC 2006)*. IEEE Computer Society, Los Alamitos, CA, USA, 408–418.

JENKS, S. F., KIM, K., LI, Y., LIU, S., ZHENG, L., KIM, M. H., YOUN, H.-Y., LEE, K. H., AND SEOL, D.-M. 2007. A middleware model supporting time-triggered message-triggered objects for standard Linux systems. *Real-Time Systems 36,* 1 (July), 75–99.

KIM, H.-J., PARK, S. H., KIM, J.-G., KIM, M.-H., AND RIM, K.-W. 2002. TMO-Linux: A Linux-based real-time operating system supporting execution of TMOs. In *Proceedings of the 5th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*. IEEE Computer Society, Los Alamitos, CA, USA, 288–294.

KIM, J.-G., KIM, M., KIM, K., AND HEU, S. 2005. TMO-eCos: An eCos-based real-time micro operating system supporting execution of a TMO structured program. In *Proceedings of the 8th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*. IEEE Computer Society, Los Alamitos, CA, USA, 182–189.

KIM, K. H. 1997. Object structures for real-time systems and simulators. *IEEE Computer 30,* 8 (August), 62–70.

KIM, K. H. 1999. Real-time object-oriented distributed software engineering and the tmo scheme. *International Journal of Software Engineering and Knowledge Engineering 9,* 2 (April), 251–276.

KIM, K. H. 2000. APIs for real-time distributed object programming. *IEEE Computer 33,* 6 (June), 72–80.

KIM, K. H. 2002. Commanding and reactive control of peripherals in the tmo programming scheme. In *Proceedings of the 5th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*. IEEE Computer Society, Los Alamitos, CA, USA, 448–456.

KIM, K. H. 2002. *Concurrency in Dependable Computing.* Kluwer Academic Publishers, Dordrecht, The Netherlands, Chapter 15: Concurrency in Dependable Real-Time Objects - Exploiting Concurrency in TMOs with Service Time Guarantees, 293–301.

KIM, K. H. 2004. Fundamental research challenges in real-time distributed computing. In *Proceedings of the 10th IEEE Int'l Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004)*. IEEE Computer Society, Los Alamitos, CA, USA, 2–9.

KIM, K. H. 2006. A non-blocking buffer mechanism for real-time event message communication. *Real-Time Systems 32,* 3 (March), 197–211.

KIM, K. H., COLMENARES, J. A., AND RIM, K.-W. 2007. Efficient adaptations of the non-blocking buffer for event message communication between real-time threads. In *Proceedings of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-time Distributed Computing (ISORC 2007)*. IEEE Computer Society, Los Alamitos, CA, USA, 29–40.

KIM, K. H., IM, C., AND ATHREYA, P. 2002. Realization of a distributed OS component for internal clock synchronization in a LAN environment. In *Proceedings of the 5th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*. IEEE Computer Society, Los Alamitos, CA, USA, 263–270.

KIM, K. H., ISHIDA, M., AND LIU, J. 1999. An efficient middleware architecture supporting time-triggered message-triggered objects and an NT-based implementation. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 1999)*. IEEE Computer Society, Los Alamitos, CA, USA, 54–63.

KIM, K. H., LI, Y., LIU, S., KIM, M. H., AND KIM, D.-H. 2005. RMMC programming model and support execution engine in the TMO programming scheme. In *Proceedings of the 8th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*. IEEE Computer Society, Los Alamitos, CA, USA, 34–43.

KIM, K. H. AND LIU, J. 2002. Going beyond deadline-driven low-level scheduling in distributed real-time computing systems. In *Design and Analysis of Distributed Embedded Systems. Proceedings of the IFIP 17th World Computer Congress. TC10 Stream Distributed and Parallel Embedded Systems (DIPES 2002)*, B. Kleinjohann, K. H. Kim, L. Kleinjohann, and A. Rettberg, Eds. Kluwer Academic Publishers, Dordrecht, The Netherlands, 205–215.

KIM, K. H., MORI, K., AND NAKANISHI, H. 1995. Realization of autonomous decentralized computing with the RTO.k object structuring scheme and the HU-DF inter-process-group communication scheme. In *Proceedings of the 2nd IEEE Int'l Symposium on Autonomous Decentralized Systems (ISADS '95)*. IEEE Computer Society, Los Alamitos, CA, USA, 305–312.

KOPETZ, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

KOPETZ, H. AND GRÜNSTEIDL, G. 1994. TTP - a protocol for fault-tolerant real-time systems. *IEEE Computer 27,* 1 (January), 14–23.

KOPETZ, H., HEXEL, R., KRÜGER, A., MILLINGER, D., NOSSAL, R., STEININGER, A., TEMPLE, C., FÜHRER, T., PALLIERER, R., AND KRUG, M. 1997. A prototype implementation of a TTP/C controller. In *SAE Congress and Exhibition*. Number 970296 in SAE Technical Papers. SAE International, Warrendale, PA, USA.

KOPETZ, H. AND OCHSENREITER, W. 1987. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers 36,* 8 (August), 933–940.

LI, Y. 2005. A model for efficient real-time distributed computing middleware incorporating a fine-grained program-segment-level deadline-based scheduling policy and an efficient checkpoint-based replication scheme. Ph.D. thesis, Department of Electrical Engineering and Computer Science. University of California, Irvine.

SHIN, I. AND LEE, I. 2004. Compositional real-time scheduling framework. In *Proceedings of the 25th IEEE Int'l Real-Time Systems Symposium (RTSS 2004)*. IEEE Computer Society, Los Alamitos, CA, USA, 57–67.

VARMA, P. 2001. Two lock-free, constant-space, multiple-(impure)-reader, single-writer structures. US Patent No. 6304924 B1.

Kwang-Hae (Kane) Kim is Professor of Computer Engineering and Computer Science at the University of California, Irvine, USA. He received an M.A. degree from the Computer Science Department at the University of Texas at Austin in 1972 and a Ph.D. degree from the Computer Science Division at the University of California, Berkeley, in 1974. He is the principal originator of the TMO (time-triggered message-triggered object, also called RTO.k) programming scheme, the Distributed Recovery Block (DRB) scheme, and several other fundamental dependable computing approaches. Dr. Kim is a recipient of the 1998 IEEE Computer Society's Technical Achievement Award and the 2004 IEEE Computer Society's Tsutomu Kanai Award for his contributions to the scientific foundation of both real-time fault-tolerant computing and real-time object-/component-oriented distributed computing. In 2007, he received the Korean Overseas Compatriots Award in the area of Science & Technology from the Korean Broadcasting System. Dr. Kim is a fellow of IEEE (elected in Fall 1988) and a member of the IFIPWG 10.4 on Dependable Computing. He served as the Chairman of the IEEE Computer Society's Technical Committee on Distributed Processing during 1984 - 1986 and hosted several IEEE conferences. He also served as a member of the founding editorial board for the IEEE Transactions on Parallel and Distributed Systems during 1989 - 1994. In 1983, he was one of the two co-founders of KOCSEA (Korean Computer Scientists & Engineers Association in America) and during 2005 - 2006, he served as the president of KSEA (Korean-American Scientists & Engineers Association).

Juan A. Colmenares received the B.Sc. degree in Electrical Engineering and the M.Sc. degree in Computer Science from the University of Zulia (Venezuela) in 1997 and 2001, respectively. He is currently a Ph.D. Candidate in the Department of Electrical Engineering and Computer Science at University of California, Irvine (USA). His research interests include real-time distributed systems, automation and control systems, and modeling and optimization. He is also a faculty member of the Applied Computing Institute, Engineering School, University of Zulia.