Programming Language Support for Implementation of Statistical Quality of Service Guarantees and Fault Tolerance

Jitendra D. Padhye and John A. Stankovic
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
e-mail: {jitu, stankovic}@cs.umass.edu

Abstract

This paper describes new programming language primitives that are designed to support statistical Quality of Service (QoS) guarantees and simple fault tolerance. We believe that such features would be useful in the implementation of next generation mechatronic systems that combine real time control applications and multimedia [GSRS96]. The paper does not describe a complete programming language, but instead builds on top of the Real-Time Concurrent C (RTCC) language described in [GR91]. Five new language constructs are introduced, each for a specific purpose. These are: specification of statistical QoS requirements, specification of complex value functions, synchronization of periodic processes, barrier synchronization of periodic processes, and simple fault tolerance via explicit task-processor binding. In each case we provide a justification for incorporating the feature into RTCC. The paper is intended as a presentation of our work in progress and discussion of possible research directions.

1 Introduction

Programming language support for real time systems has long been an active area of research. Several programming languages such as Euclid [KS86], Flex [KL91], Real-Time Concurrent C [GR91] and Real-Time C++ [ITM90] have been described in the literature. Many of these languages are designed to address the issues faced in *traditional* real time applications. These languages provide constructs for specifying deadlines and periods of tasks, bounded-waiting IPC primitives and critical regions. However, most of these languages lack features to support statistical Quality of Service

(QoS) guarantees and complex value functions to describe post-deadline behavior. We believe that such features would be useful in the implementation of next generation mechatronic systems that combine real time control applications and multimedia [GSRS96], [GPLRS95]. This paper is an attempt to describe a minimal but general set of language constructs that can be incorporated in the Real-Time Concurrent C (RTCC) language described in [GR91], to satisfy these needs. In addition, we also describe a programming language primitive that can assign tasks to processors and demonstrate how simple fault tolerance can be achieved by using it in conjunction with bounded-waiting IPC calls provided by RTCC. Mechanisms for adaptive fault tolerance are being further explored by our group [GSR96]. This paper is intended as a presentation of our work in progress and discussion of possible research directions.

The rest of the paper is organized as follows. In Section 2, we briefly describe the relevant features of the RTCC language. Section 3 summarizes the requirements of a modern control system, which provide motivation for this paper. Section 4 presents the new, proposed language features, and describes each one in detail. Section 5 describes some related work in the area. Section 6 concludes the paper and describes avenues for future work.

2 Real Time Concurrent C

Many mechatronic applications are implemented using C. However, C itself does not contain features for supporting the real time aspects of mechatronic systems. Real Time Concurrent C adds such features. However, RTCC had no features for supporting multimedia. By adding real time and multimedia features to C, we can build upon a widely used programming language. In this section we briefly describe several relevant features of RTCC. More details can be found in [GR91].

- **Activities with Deadlines:**

  The language allows timeouts to be associated with any statement using the `within deadline` construct. If the execution of the statement is not completed within the given time limit, the execution is aborted. For example, the following describes the scenario where a pilot has to get landing clearance from airport control in 60 seconds, failing which the landing must be aborted.

  `within deadline` (60) *get clearance*;

  `else` *abort landing*

In addition, the language supports specification of *slop*, which is used to specify the additional time interval *after the deadline* for which the activity may be allowed to continue, should it fail to terminate by its deadline. For example the following describes a scenario where a clearance must be obtained, preferably in 60 seconds, failing which, there is a 10 second grace period, after which the landing attempt is aborted.

`within deadline` (60)(10)*get clearance*;

`else` *abort landing*

- **Periodic Activities:**

  Periodic activities can be specified with following construct:

  `every` (p) `[ until` *condition* `]`

      $statement_1$

  `[ else` $statement_2$ `]`

  If the control reaches the `every` statement at time t, $statement_1$ will be executed once every p seconds until *condition* becomes true. The i^{th} execution will begin at time $t + (i - 1) * p$. If execution of $statement_1$ is not completed within its period, $statement_2$ is executed. The following example describes a process control application in which the temperature is read every second until a shutdown is specified. If, for some reason, the reading cannot be taken within 1 second, notify the operator:

  `every` (1) `until` *shutdown*

      read temperature

  `else` *notify operator*

- **Guaranteed Activities:**

  This construct is used to ensure that an activity will be started if and only if the scheduler can guarantee that it will finish by its deadline. Note that the activity itself must be time constrained in nature. Consider the following construct:

  `within deadline` (gd) `guarantee`

      `within deadline` (d)

          $statement_1$

      `else` $statement_2$

  `else` $statement_3$

Let us assume that control reaches the `guarantee` statement at time t. Then, $statement_1$ will start executing if and only if, the scheduler can guarantee before time $t + gd$ that $statement_1$ will finish in time d once it starts. If the scheduler can not guarantee this, (say, due to unavailability of resources needed by $statement_1$), then $statement_3$ starts executing. If $statement_1$ starts executing (say, at time t_1) and fails to finish by time $t_1 + d$, (say, due to an error condition), its execution will be aborted and $statement_2$ will start executing. Note that in this example, the `guarantee` construct imposes a timing restriction on the scheduler, not on the task itself.

3 Requirements of Next Generation Control Systems

Conventional control applications rarely rely on audio and video data to perform their control functions. However, as high bandwidth networking, processing power and physical memory become cheaper, incorporation of multimedia technology into control applications is becoming feasible. Control applications can benefit from multimedia technologies in many different ways. The authors of [GSRS96] suggest that the technology can be used at two different levels – operator and management. At the operator level, the multimedia (audio and video) technology could be used to enhance safety and confirm readings made by other, conventional sensors. It can also be used for maintenance (ultrasound imaging to detect deposit buildups in fluid pipes). At the management level, technology can be used for *post-mortem* analysis of equipment failure or other catastrophic events. The multimedia technology can also be used in planning and design. Live images from the plant floor are used to create a virtual-reality, 3-D map of the plant to aid in design and process modifications. In [GPLRS95], the authors describe a prototype control system that uses infra-red video data to detect a flaw in a coating process and sound an alarm when appropriate.

In any control system that uses multimedia data, either in the control loop itself or outside of it, managing the end-to-end QoS becomes an important issue. This is achieved by appropriate scheduling techniques. We propose a language construct which allows multimedia tasks to declare their jitter tolerances, and show how a scheduling algorithm might use this information to achieve higher throughput. Apart from maintaining QoS of individual streams, it may also be important to keep two or more data streams synchronized within specified limits. We describe two language constructs aimed towards specifying such constraints. In large, flexible manufacturing systems, the relationship between the completion time of a task and the value accrued by it can be quite elaborate. We describe a way by which tasks can make this relationship known to the scheduler. The scheduler can then use this information to try to come up with a schedule that maximizes the accrued value. Finally, we describe a simple construct that facilitates implementation of fault tolerance techniques in a multi-processor environment.

4 Proposed Language Constructs

Each construct in this section is described in a separate subsection. For each construct, we initially provide a justification for incorporating that construct in the language. This is followed by the syntactic and semantic description of the construct. The usage of each construct is then demonstrated by a small example.

4.1 Statistical Guarantees

Most multimedia tasks do not require the type of hard guarantees required by traditional real-time tasks. Multimedia tasks tend to be periodic in nature (e.g., process and display frames of video at the rate of 30 frames per second) and can tolerate a certain amount of *jitter*. The term *jitter* is defined in several different ways in the literature. For the video-frame example mentioned above, jitter can be the variance in the frame rate. In terms of scheduling requirements, this jitter tolerance can be expressed as a two-tuple (x, y), where x denotes the percentage of total invocations of the periodic tasks that miss their deadline, while y denotes the maximum percentage lateness of these tasks. The scheduler can take advantage of this information to serve more tasks. This is illustrated in the following example:

Consider a video being displayed at the rate of 25 frames per second. Assume that the work is being done by a task that is invoked 25 times a second (i.e., period of the task is 40 milliseconds.). Let the worst case execution time (WCET) of the task be 1 millisecond and its deadline be 10 milliseconds. It is easy to see that no more than 10 such *simultaneously arriving* tasks can be supported on a uniprocessor system.

Now consider that 11 such tasks arrive *simultaneously* and each of them has specified its jitter requirements as $(10, 10)$. This means that for each task, 10% of its invocations (remember that each task is invoked every 40 milliseconds) can miss their deadlines, and they can miss them by at most 10%. In other words, 10% invocations of each task can be considered to have a deadline of 11 milliseconds. With this information, the scheduler can create a schedule where each task meets

```
within deadline (20) guarantee (10, 10)
{
        every (40)
        {
                within deadline (10) display_next_frame ;
        }
        else abort ;
}
else abort ;
```

Figure 1: The extended `guarantee` construct

its deadline within the specified jitter constraints. At the beginning of every 40 millisecond period, the scheduler randomly selects a task to miss the deadline. Each task has an equal probability of being selected. The scheduler then schedules the remaining 10 tasks in any order in the first 10 milliseconds of the 40 millisecond period and the remaining "late" task is scheduled during the 11^{th} millisecond. Note that the scheduler makes use of this "jitter relaxation" only when it is absolutely necessary. For example, if we had only five (instead of 11) tasks in the above scenario, the scheduler will simply ignore the jitter related information[1].

We augment the `guarantee` primitive provided by RTCC to include the jitter information as discussed above. Needless to say, this information makes sense if and only if the guarantee is being sought for a periodic task. The new `guarantee` primitive is:

```
        within deadline (gd) guarantee (percent_miss, percent_max_late)
                .
                <definition of periodic task>
                .
        else [error-handle]
```

Figure 1 shows an example program that implements the video task discussed above using this construct.

[1]This example is not meant only to demonstrate how jitter related information can be used for scheduling purposes. We are not proposing a new scheduling algorithm.

4.2 Synchronization of Periodic Tasks

Synchronization between multiple media streams is an important concern in developing a large multimedia application. Consider, for example, an audio-visual presentation demonstrating a certain product. If the targeted audience is multi-lingual, it is necessary to store the audio streams in various languages separately from the video stream. During presentation, the video stream and the audio stream requested by each user must be processed in a co-ordinated manner to ensure meaningful delivery of the presentation.

The need for synchronization between two periodic tasks can be a requirement of hard real time applications as well. Consider a chemical plant monitoring system that is monitoring a very rapid and possibly violent chemical reaction. The process can consists of periodically adding reactants into the reaction vessel. If the flow of different reactants is controlled by separate periodic processes, it might be necessary to synchronize the execution of these two periodic tasks.

For the purpose of this paper, we restrict our definition of synchronization to tasks with the same period. The definition can be extended, but it does not change the basic constructs introduced here. We consider two periodic tasks P_i and P_j to be synchronized within time δ of each other, if for each k, the k^{th} invocation of task P_j starts within time δ after the k^{th} invocation of task P_i finishes execution, or vice-versa. The exact order is controlled by keywords `AFTER` or `BEFORE`[2].

We introduce several new keywords to specify synchronization. The keyword `TASKNAME`, which optionally follows the `every` keyword, is used to label a task. An optional statement following the `every` keyword defines synchronization details. It is of the form: `SYNCHRONIZE <time> BEFORE|AFTER <task name>`. The meaning of this syntax should be self evident.

Let us now examine the semantics of the `SYNCHRONIZE` statement with the help of the example shown in Figure 2. The code segment essentially says that the audio segment corresponding to a video frame needs to start playing within 0.5 milliseconds of the frame being displayed. The audio error handler can execute if either the segment misses its 40 millisecond deadline (implied by the period) or if the synchronization is missed. The audio error handler does not execute if the video task somehow terminates. In other words, the `SYNCHRONIZE` stops having any effect if the task it is synchronizing *WITH* terminates. This, however, does not signal an error.

[2]In addition, we can introduce the the concept of *synchronization jitter*, which is similar to scheduling jitter described previously. The jitter is specified by a two-tuple (x, y) where x specifies the percentage of invocations that can miss the jitter constraint and y indicates the maximum percentage by which this constraint may be violated.

```
every (40) until (VIDEO_OVER == TRUE) TASKNAME=VIDEO
        <display next frame>
else <execute video_error_handler>

every (40) until (AUDIO_OVER == TRUE) TASKNAME=AUDIO
SYNCHRONIZE 0.5 AFTER VIDEO
        <play the next audio segment>
else <execute audio_error_handler>
```

Figure 2: A synchronization example

4.3 Barrier Synchronization

The need for barrier synchronization has been long recognized and fulfilled in the parallel programming community. Sophisticated barrier synchronization primitives, like those described in [PICL] have been developed and analyzed. We introduce a very simple version here [3]. Our synchronization primitive allows for a one-time synchronization of two periodic processes. The processes need not have the same period; indeed, their periods can be relatively prime. As a motivation for introducing this language construct, consider an audio and a video being displayed simultaneously for a user. The video stream can be a time-lapsed-photographic sequence of a slow process (such as slow corrosion of a pipe) while the audio can be a general lecture on the subject of corrosion. We do not care for exact synchronization between these two streams because of the following reasons:

- The two streams are being displayed at different rates.
- There is no immediate correspondence between the audio and the video, i.e., the speaker is not describing the image, but only its general content.

We do, however, want to ensure that when pictures of water induced corrosion are being shown, the audio streams are not describing the air-induced corrosion. In other words, from time to time we want to "stop" the audio to let the slower moving video catch up. This can be accomplished via two shared variables between the audio and the video processes and a barrier synchronization primitive. Let us assume that the video sequence takes 25 frames to display water-induced corrosion. Let us also assume that the description for each type of corrosion is done in approximately 20 invocations

[3] To be precise, this construct is semantically a little different from the barrier synchronization process described in [PICL]. It, however, achieves a similar purpose.

```
every (5) TASKNAME=A BARR_SYNCH=a_count
{
        static int i=0 ;
        if ( (i > 10) && (i%10 == 0) )
                BARRIER B 14
        else {
                i ++ ;
                <execute task A>
        }
}
else <execute err_handler>

every (6) TASKNAME=B BARR_SYNCH=b_count
{
        <execute task B>
        b_count = (b_count+1)%15 ;
}
else <execute err_handler>
```

Figure 3: Example of Barrier Synchronization

of the audio task. To maintain synchronization, the audio stream checks to see if 25 video frames have been displayed at the end of every 20 invocations. If fewer frames have been displayed, it simply skips execution until the video catches up. This is accomplished by using the `BARRIER` primitive as follows.

Each periodic process defines a variable of type `BARR_SYNCH`. A task can specify `BARRIER` at any time during its execution. The task specifies the name of the other task and a value. Once a `BARRIER` statement is executed, the task finishes executing its current invocation normally. However, from the next invocation onwards, the task executes if and only if the value of the `BARR_SYNCH` variable in the specified task is greater than or equal to the barrier value specified by the waiting task. Figure 3 shows two periodic tasks. After every 10 invocations, task A waits until task B has managed to finish 15 invocations.

4.4 Complex Value Functions

RTCC provides no support for specifying the *value* of a periodic or sporadic task. Also, it provides very little support for specifying *soft* deadlines. The need for complex post-deadline behavior arises when an application that has missed its deadline accrues a different value depending on its actual finish time.

Examples of complex post-deadline behavior are easy to find in manufacturing scenarios. Consider a situation where the controller of a flexible manufacturing facility is trying to schedule production orders from several customers. Depending upon the guarantees given to each customer and the terms of contracts (what happens if the promised deadline is not met), it may be advisable to delay orders from certain customers deliberately, to achieve higher profits. This, however, means that the value function (relationship between finished time and accrued value) must be available to the scheduler.

We extend the `within deadline` construct in RTCC to provide for a complex value function as follows:

```
within deadline (d) guarantee
        within deadline (gd) (slop) (value_function)
                <task definition>
        [else statement2]


double value_function (end_time, gd)
{
        <return value based on curr_time and gd>
}
```

In the above piece of pseudo-code, `(value_function)` provides a function pointer to the function that computes the value function[4]. It is important to note that the function is used by the scheduler and *not* by the user program itself[5].

We assume a scheduling paradigm in which the arrival of new tasks does not violate the guarantees given to previous tasks. If the new task can not be guaranteed completion by its requested deadline, the scheduler computes the expected finish time of the task and uses the value function to determine the expected value. Based on this value, the scheduler may choose to accept or reject the new task.

[4]Since RTCC is a superset of C, we assume that the name of a function serves as a pointer to that function.

[5]Prof. Krithi Ramamritham has suggested that the function can be used by the application program itself if it is allowed to schedule its component functions e.g. while using real-time threads.

```
within deadline (3) guarantee
        within deadline (5) (10) (val_func)
                do_specified_task ;
        else call_error_handler

val_func (time, deadline)
{
        if time < deadline
                return 10 ;
        else if time < deadline+10
                return -1*time+deadline+10 ;
             else
                return 0 ;
}
```

Figure 4: Use of Value Function Specification

During actual execution, if the task does not finish execution by $start\_time+gd+slop$, the execution is interrupted and `statement2` is executed. In that case, the actual value accrued by the schedule depends on the value specification of `statement2`. An example program in Figure 4 shows how the value function can be used to signify that the value of the task decreases gradually from 10 to 0 within 10 time units after the deadline.

4.5 Fault Tolerance

Since most real time systems are designed to perform critical tasks, fault tolerance is an important design requirement for most of them. Language features that make programming of fault tolerance strategies easier are necessary to facilitate the design of such systems. However, it is important to realize that predictability can not be sacrificed for implementing elaborate fault tolerance mechanisms. With this constraint in mind, we describe a single additional construct for RTCC that allows implementation of simple fault tolerance strategies.

Simple fault tolerance strategies such as triple modular redundancy or primary-backup can be used to achieve limited hardware fault tolerance if specific tasks can be assigned to run on specific processors. To achieve this task-to-processor binding, we introduce the primitive `ON_PROC`. We assume that the underlying hardware has a MIMD architecture. The `ON_PROC` keyword can be used after any `TASKNAME` definition to assign that task to a particular processor. Whenever such a binding is specified, at run time the scheduler assigns the task to the processor to which it has been bound.

Figures 5 and 6 show an example of a simple program that uses asynchronous IPC and the `ON_PROC` construct to implement a triple modular redundancy fault tolerance paradigm. The controller task, which executes on the system processor P0, waits for 10 time units to receive a result from any of the three identical work tasks, which are executed on three different application processors. The work tasks themselves have a deadline of 10 time units each[6]. Depending on the majority, a particular result is chosen as the final result and the dissenting processor, if any, is declared faulty. Note that by using asynchronous communication primitives, we have managed to keep the execution time of the controller predictable, even if the application processors were to fail completely.

5 Related Work

The Real Time Concurrent C language (RTCC), which forms the basis of this paper, was proposed by Gehani and Ramamritham in [GR91]. The language is an extension of the Concurrent C language. The language provides facilities (some of which have been described previously in this paper) to support description of periodic and non-periodic tasks and deadline constraints. It also allows applications to seek timing guarantees and perform alternative actions either when deadlines are not met, or when timing guarantees can not be given. The language, however, does not provide support for describing statistical QoS constraints, mutual timing constraints between multiple periodic tasks and support for specifying complex value functions. No support for specifying fault tolerance is provided either.

The FLEX language is described in [KL91]. The language is based on the concept of imprecise computations, allowing for the tradeoff between performance and quality. FLEX also provides support for describing complex timing relationship between tasks. However, it lacks features for specifying statistical QoS guarantees and complex value functions. Another significant drawback of the language is that the timing guarantees provided by the language may not hold under worst case assumption.

In [ITM90] the authors describe the Real Time C++ language. The language builds on the

[6]The task code should contain an asynchronous message send somewhere, but these details have been omitted.

```
within deadline (5)  guarantee
        within deadline (20) TASKNAME=CONTROL ON_PROC=P0 {
                time = clock () ;
                res1 = res2 = res3 = INVALID_VALUE ;
                while (clock()-time < 10) {
                /* from task T1 on proc P1 receive a message of type RESULT */
                        res1 = asynch_recv (T1, P1, RESULT) ;
                        res2 = asynch_recv (T2, P2, RESULT);
                        res3 = asynch_recv (T3, P3, RESULT);
                }
                if ( (res1 == INVALID_VALUE) &&
                     (res2 == INVALID_VALUE) &&
                     (res1 == INVALID_VALUE) ) {
                        <execute error_handler>
                       exit ;
                }
                else if ( (res1 == res2) && (res2 != res3) ) {
                         final_result = res1 ;
                         <processor P3 declared bad>
                     }
                     else if ( (res1 == res3) && (res3 != res2) ) {
                              final_result = res1 ;
                              <processor P2 declared bad>
                          }
                          else if ( (res2 == res3) && (res3 != res1) ) {
                                  final_result = res2 ;
                                  <processor P1 declared bad>
                               }
                               else if ( (res1 == res2) && (res2 = res3) )
                                        final_result = res 1 ;
                                    else {
                                         /* all results disagree */
                                         <call error handler>
                                    }
        }
```

Figure 5: Fault Tolerance using ON_PROC statement: Controller Task

```
within deadline (5) guarantee
        within deadline (10) TASKNAME=T1 ON_PROC P1
                <task 1 code>

within deadline (5) guarantee
        within deadline (10) TASKNAME=T2 ON_PROC P2
                <task 2 code>

within deadline (5) guarantee
        within deadline (10) TASKNAME=T3 ON_PROC P3
                <task 3 code>
```

Figure 6: Fault Tolerance using ON_PROC statement: Work Tasks

object-oriented nature of C++. Timing constraints can be attached to both objects as well as specific operations performed on the objects. The language is built around the rate-monotonic scheduling paradigm. Guarantees are provided only to periodic processes and for aperiodic code segments, exception handling is provided when deadlines are missed. The language provides no support to describe statistical QoS constraints.

In [SWM95], the authors describe a data model for multimedia presentations that allows formal specification of QoS constraints and based on an ideal presentation output. The authors also present an error model and show how it allows better resource optimization. The authors, however, do not discuss a procedural programming language implementation of their work.

A general framework for adaptive fault tolerance is proposed in [BSS95]. The authors describe syntax and semantics of FERT (Fault Tolerant Entities for Real Time), which is presented as an intermediate level structure between system design and application code. The structure provides for explicit specification of real-time constraints and flexible and adaptable control strategy for managing redundancy within software application modules. FERT, however, by definition, is not a programming language feature.

6 Conclusions and Future Work

We have presented a total of five new language constructs for RTCC which provide the following features:

- Specification of statistical QoS requirements
- Specification of complex value functions
- Synchronization of periodic processes
- Barrier synchronization of periodic processes
- Simple fault tolerance via explicit task-processor binding

We have described how each of these features might be of use in design of modern control systems. It can be argued that many of these features can either be provided as system calls (e.g., specification of jitter constraints), or can be incorporated by explicit programming (e.g., implementation of barrier synchronization algorithm in application code). We believe that by providing these features as an integral part of the programming language enhances their ease of use. This work is preliminary in nature and serves to demonstrate the need of such features in any language used to implement a modern control system. A more detailed analysis of each feature is required to warrant its inclusion in an actual implementation of RTCC. Such an implementation will provide more insight about how information provided by such features can be used by the scheduler to achieve better performance.

References

[BSS95] A. Bondavalli, J. Stankovic and L. Stringini, "Adaptable Fault Tolerance for Real Time Systems," *Responsive Computer Systems: Steps Toward Fault Tolerant Real-Time Systems*, Kluwer Academic Publishers, 1993, pp. 187-208.

[GPLRS95] A. Guha, A. Pavan, J. Liu, A. Rastogi and T. Stevents, "Supporting Real-Time and Multimedia Applications on Mercuri Testbed," *IEEE Journal on Selected Areas in Communications*, Vol.13, No. 4, May 1995.

[GR91] N. Gehani and K. Ramamritham, "Real-Time Concurrent C: A Language for Programming Dynamic Real Time Systems," *J. Real Time Systems*, Vol. 3, No. 4, December 1991, pp. 377-405.

[GSR96] O. Gonzalez, J. Stankovic and K. Ramamritham, "Support for Adaptive Fault Tolerance on the Spring Real-Time System," *Technical Report Department of Computer Science, University of Massachusetts*, 1996.

[GSRS96] O. Gonzalez, S. Sen, K. Ramamritham and J. Stankovic, "Incorporation of Multimedia Capabilities in Distributed Real-Time Applications," *To appear in Workshop on Databases: Active and Real-Time (Concepts meet Practice)* to be held in Rockville, Maryland November 15, 1996.

[ITM90] Y. Ishikawa, H. Tokuda and C. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," *Proc. ECOOP/OOPSLA '90*, October 1990, pp. 289-298.

[KL91] K. Kenny and K. Lin, "Building Flexible Real Time Systems using the Flex language," *IEEE Computer*, Vol. 24, No. 5, May 1991, pp. 70-78.

[KS86] E. Kligerman and A. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986, pp. 941-949.

[PICL] G. Geist, M. Heath, B. Peyton and P. Worley, "PICL, A PORTABLE INSTRUMENTED COMMUNICATION LIBRARY, C REFERENCE MANUAL," Oak Ridge National Laboratory, 1990.

[SR91] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real Time Systems," *IEEE Computer*, May 1991, pp. 62-72.

[SWM95] Richard Staehli, Jonathan Walpole and David Maier, "Quality of Service Specification for Multimedia Presentations," *Multimedia Systems*, Vol. 3, number 5/6, November 1995.