MESSAGE COMMUNICATIONS IN A DISTRIBUTED REAL-TIME SYSTEM WITH A POLLED BUS*

Kang G. Shin   Yogesh Muthuswamy

Real–Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109–2122.

ABSTRACT

This paper addresses high-speed message communications in a distributed real-time system. Each node in the system consists of multiple processors interconnected by a broadcast bus, but the individual nodes are interconnected by an arbitrary network.

The concept of a *poll number* is proposed to control the access to an intra-node bus. The bus access mechanism with the poll number is intended to minimize the probability of real-time messages missing their deadlines. When a task generates a time-constrained message, a poll number associated with the message is computed based on the message's deadline and the task's priority. When the bus is free, the various tasks at a node which desire to use the bus write the poll number onto the bus and read it back, one bit at a time, starting from the most significant bit. If at any time the bit read back is different from the bit written, then that particular task drops out of the competition for the bus.

The above access mechanism provides for not only decentralized control of the intra-node bus, but also a high degree of flexibility in scheduling messages via different ways of generating poll numbers. The performance of the polled bus is analyzed and then compared with that of a token bus, which is widely used in computer-controlled systems, e.g., computer-integrated manufacturing systems. The probability of a message missing its deadline in a token bus is found to be much higher than that of the polled bus.

1 Introduction

The most important requirement of a real-time system is to complete time-critical tasks before their deadlines. The probability of a task missing its deadline, called the *probability of dynamic failure* in [1], is thus an important performance parameter of any real-time system. This probability is strongly dependent on the mechanisms used to implement inter-task communication. The work described in this paper is primarily targeted at minimizing the probability of a task missing its deadline by speeding up the inter-task communication.

Due to its potential for high performance and reliability with multiplicity of processors, a distributed system is a natural candidate for implementing real-time applications. Each node in the system is assumed to consist of several processors connected via a simple, but high-speed, communication medium, called the *polled bus*. Each

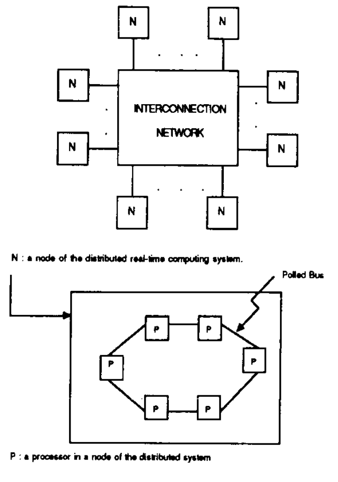

N
INTERCONNECTION NETWORK
N : a node of the distributed real-time computing system.
Polled Bus
P
P : a processor in a node of the distributed system

Figure 1: A model of a distributed system

processor at a node executes one or more tasks, realizing some useful, probably time-constrained, function, e.g., controlling a robot in a workcell node. It is often necessary that the tasks, within a node as well on different nodes, exchange information to realize the overall functions of the system.

The system model is depicted in Fig. 1. The model is similar to the one described in [2], where the processors within a node are connected by a token bus [3]. The main difference between a token bus and the polled bus lies in their access protocols. Our system may be used for real-time industrial control applications such as control of a totally automated manufacturing shop, where all the functions are done by machines with little or no human supervision. The shop floor may consist of several workcells that have many devices which coordinate to perform a useful industrial function. Each workcell may be controlled by a node computer system, and the devices within a workcell are controlled by the processors in the node. The tasks which control the devices within a workcell typically have stringent time constraints. All the processors which control the devices within a workcell are placed on a single board along with a communication processor (CP). The CP enables inter-task communication within a node as well as across node boundaries. One such board represents

*The work reported in this paper was supported in part by the National Science Foundation under grant No. DMC–8721492. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the NSF.

a node of the distributed real-time system.

Token bus and the polled bus may be categorized as *broadcast buses*[1]. We do not restrict the type of interconnection between the various processors as long as it has the broadcasting capability. Communication across node boundaries does not usually have any real-time constraints. Typically, the nodes may be connected by a token ring local area network. The tasks communicate with each other by message passing [4]. A protocol such as the GM's MAP [5] may be used for communication *across* node boundaries. One of the CP's functions is to provide the protocol support for implementing communications between any two tasks, which may either be in the same node or in different nodes.

The tasks within a workcell node are responsible for real-time devices such as robots and sensors. The tasks for controlling and sensing these devices are inherently periodic; e.g., the task for closing a digital servo control loop may be executed once every 100 ms and the task for sensing and analyzing parts on a conveyor belt may have to be executed once every 0.5 second for visual servoing. Aperiodic tasks, albeit infrequent, also exist within a workcell, e.g., operator's commands in response to abnormal circumstances. Periodic tasks are "base load", whereas aperiodic tasks are "random disturbances". The main intent of this paper is to deal with the "base load" or periodic tasks; treatment of aperiodic tasks is usually formulated as a dynamic load sharing problem, which is the subject of a future paper.

The bus access mechanism used in the token bus of [6] is the token ring protocol, where a single token goes around the ring. Any processor on the ring desirous of using the bus should capture the ring and release it upon completion of its service. This mechanism distributes access in a round robin manner, wherein higher priority tasks (processors) might be forced to wait longer than necessary to procure the bus while the lower priority tasks are accessing the bus. This problem may partially be solved by providing multiple priorities within the token passing bus scheme. In the token passing bus method, the *class of service* mechanism can be used to provide prioritized (4 levels) access [7]. The token bus access mechanism belongs to a class of Controlled Demand-Adaptive Multiple Access Protocols [8]. Priority access schemes using CSMA-CD for multiple priority message classes have also been proposed and analyzed [9]. According to [8], these schemes belong to the class of Contention-based Multiple Access Protocols.

The Controlled Demand-Adaptive Multiple Access Protocols can be broadly divided into token-passing and reservation schemes [8]. The token bus is an example of a token-passing scheme. The reservation scheme involves a reservation period which is divided into slots, wherein all stations (processors) which have messages to transmit post their reservation by transmitting a burst of noise during their assigned slot. After the completion of the reservation period, a station is selected based on a predetermined scheme known to all stations. The reservation scheduling protocol (RSP) proposed in [10] falls into this reservation scheme category. The RSP scheme requires a reservation period, during which the the highest priority message is selected (irrespective of the station it originated from), followed by a scheduling period, during which the station is selected based on some scheme (e.g., round robin). However, there are two drawbacks associated with the RSP protocol. First, it requires two sequential steps for reservation and scheduling, which induce a longer delay in message delivery than using only one short combined step. Second, the scheduling period and the selection schemes of the RSP are senstive to the number of stations using the channel.

[1]Any bus structure wherein the participants need not be aware of the other users of the bus.

To remedy the drawbacks of the RSP and a token ring bus, and thus, to minimize the probability of a task missing its deadline, we propose a new bus access mechanism, which is somewhat similar to the one used in FTMP [11]. Unlike the RSP protocol [10], our scheme combines the reservation and scheduling periods into a single *polling round.* It ensures that at the end of the polling round, only one station (processor) will have control of the bus.

In our scheme, each processor computes a *poll number* which is determined by various factors, including the deadline and user-assigned priorities. When a task generates a message service request, it waits for the bus to complete the present request and enters a polling round together with the other processors wanting to use the bus. After a fixed amount of time (polling time), the processor with the highest poll number is guaranteed to get control of the bus. This proves to be superior to the token bus since it gives greater priority to tasks with closer deadlines, amongst other factors. This scheme is better than the RSP, since the polling round is not dependent on the number of processors. In our scheme, each processor need not be aware of the other processors' priorities, nor of the number of such processors contending for the bus.

Because of the nature of the bus, we refer to the intra-node bus as a *polled bus* in contrast to the token bus of [6]. The proposed bus access mechanism also provides a decentralized bus control mechanism with predictable and optimum performance parameters.

This paper is organized as follows. The polled bus access mechanism and the poll number design, along with the benefits of using the poll number, are discussed in Section 2. In Section 3, the performance of the polled bus and the token bus approaches are analyzed and compared. The paper concludes with Section 4.

2 The Polled Bus Access Mechanism

Intra-node communications are constrained by the real-time requirements of the various tasks at the node. The design of the interconnection mechanism for processors within the same node has to satisfy the real-time constraints imposed by the various tasks. The interconnection mechanism (bus) may be controlled by a dedicated processor which is central to the node. This processor could then decide on the allocation of the bus to the various tasks so that the probability of missing deadlines is minimized. However, the failure of the central control processor would paralyze the communications within a node leading to a potentially disastrous situation. Hence, we propose a bus access mechanism which provides for decentralized control of the bus.

As mentioned earlier, the processors in a node are interconnected by a broadcast bus, which enables the processors to read from and write into the bus without being aware of the presence of other processors. Typical examples are time-shared unibus, token ring, token bus, etc. The general structure of a node is shown in Fig. 2. The software executing on each processor may be partitioned into device control and interface software. Among the important functions of the interface software is bus access. The software implements the decentralized bus access algorithm which we will describe in the next section.

In [6], a token bus interconnects the various processors within

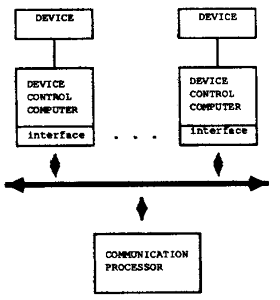


Figure 2: The structure of a single node.

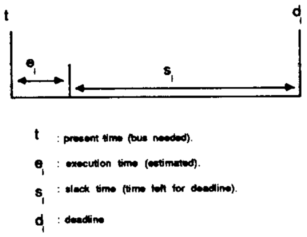

t
d_i
e_i
s_i
t : present time (bus needed).
e_i : execution time (estimated).
s_i : slack time (time left for deadline).
d_i : deadline

Figure 3: Defining the "slack time".

a workcell node. In a token bus interconnection scheme, a token circulates around the ring. The processor which needs to send a message using the bus should capture the token to control the bus and release the token for circulation as soon as it sends the message. The token travels to the logically adjacent processor on the bus, as in the case of a token ring.

Consider the four processors P_1, P_2, P_3 and P_4 connected to the token bus. Assume that the priorities of the tasks are such that $Pr_1 \leq Pr_2 \leq Pr_3 \leq Pr_4$, where Pr_i represents the priority of the task executing on processor i. This means that P_4 executes the most critical task(s). If P_4 sends a message and then relinquishes the bus (token) to the next processor downstream, i.e., P_1, then it has to wait the whole round before sending the next message (if any). The best case scenario is that none of P_1, P_2, and P_3 have any message to send. The worst case occurs when all the other three of them have messages to send. In this case, the task executing on P_4 may get delayed and may even miss the deadline, which might prove catastrophic if it has to execute a very critical task. From the above discussion, it is easy to see that the task with the highest priority among the tasks competing for the bus should be given control of the bus. These priorities may be user specified, or in the absence of any user specification may be assumed to be the time left for the lapse of the deadline. In our proposed mechanism the above scenario will not occur since the task whose message has the closest deadline and/or the highest priority will get the bus before any other task.

2.1 Bus Acquisition Algorithm

A high speed bus interconnects the various processors of a node. It is assumed that the processors can detect if the bus is busy. This may be accomplished by the use of a bus busy line. The logical structure of the bus is circular. Whenever the bus is in use (for sending messages) a line (busy line) is set high. All the processors connected to the bus can sample this line.

When a processor needs to use the bus, it first samples the busy line. If the bus is busy, the processor busy-waits for the bus. As soon as the bus is free, the processor attempts to acquire the bus by initiating the acquisition process. Any number of processors might try to acquire the bus simultaneously. Each processor computes a number (which should be unique) called the *poll number*, which consists of a finite number of bits (say m bits). The number is divided into several fields, with each field having a special significance. The determination of the exact structure of the poll number and its design will be discussed in the next subsection. This number is guaranteed to be unique, i.e., no two processors can produce the same poll number. This poll number is so designed that the task whose message has the closest deadline will have the largest poll number. The bus performs a wired-OR operation on all the signals impinging on it from the various processors. A processor competing for the bus writes the poll number to the bus, one bit at a time, starting with the most significant bit. After writing each bit, it waits for a finite interval[2] and samples the bus. If the value read by the processor is different from the value it wrote into the bus, it drops out of contention for the bus. This situation will arise only when a processor with a larger poll number is contending for the bus. After m such rounds, the processor with the highest poll number has sole control of the bus.

2.2 Design of the Poll Number

The poll number is computed by every task that needs to use the bus. The design of the poll number acutely affects the performance of the system. The algorithm does not require any state exchange between processors thereby obviating the need for maintaining the global system state at each node.

The main issues in the design of the poll number are :

1. The number of bits, m, used to encode the poll number and how to ensure its uniqueness without sacrificing other information.

2. The significance and the ordering of the various fields of the poll number so as to minimize the number of messages missing their deadlines.

First, we shall determine the various fields of the poll number and their relative ordering. A *field* is a contiguous collection of bits which has a special significance. A field, called the *uniqueness* field, assigns a unique identification number to each of the processors in a node so that no two poll numbers will be identical in any situation. Another field, called the *deadline* field, is necessary to represent the slack time (the time left until the deadline) or some encoding of it. (See Fig. 3 for the definition of the slack time.) The user defined priorities of the tasks is defined in the *priority* field. It is easily seen that the uniqueness field design depends on the number of processors and the priority field is related to the tasks which resides in these processors.

The ordering of these fields depends on the system design objectives. In our design we would like to minimize the number of messages missing their deadlines. Hence, the *most significant* field is the deadline field in which each message's deadline is encoded

[2]To propagate and stablize the bit written on the bus. This is a function of the physical length of the bus.

Figure 4: Field structure of the poll number.

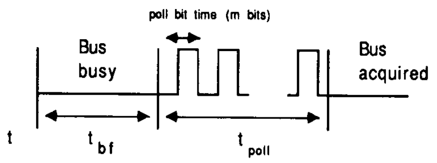


Figure 5: A typical bus access cycle in a polled bus.

to be *inversely* proportional to its actual deadline. The next significant field is the priority field. In a situation where two competing tasks have the same deadline, the task with the greater user/system assigned priority gets the bus. The least significant is the uniqueness field, which comes into play only when competing processors have identical deadline and priority fields. In order to avoid biasing the bus access towards any one processor on account of the uniqueness number, the processor identification numbers which form the unique ness field component can be assigned in a round robin fashion. The ordering of the three fields of the poll number is shown in Fig. 4.

The number of bits in the poll number, m, will now be determined We shall first do a simple analysis and obtain an expression for m. Then we shall show with the help of typical numerical values that the value obtained for m is far in excess of what is actually required. A simple encoding[3] is used to obtain the poll number, in particul: the deadline field.

Let N_{proc} be the number of processors at a node, and N_{tasks} be the number of tasks resident at the node. (If only a single task resid·s at a single processor, then $N_{proc} = N_{tasks}$.) The number of bits needed in the uniqueness field is $log_2\ N_{proc}$ and the number of bits needed for the priority field is $log_2\ N_{tasks}$. Unlike the uniqueness and priority fields, determination of the maximum number of bits neede' to represent the deadline field is not so straightforward.

Consider Fig. 5 which shows a bus access cycle. At time t, the processor (or task executing on that processor) generates a messag· and thus requires the bus to send it. The time interval between the generation of messages is assumed to be exponentially distributed. The processor waits for a time t_{bf} for the bus to become free. Then, it enters the *polling round*, irrespective of whether it has any competitor or not. The polling round consists of m poll bit times, where each *poll bit time* consists of a write, an interval to stablize the bit written on the bus, and a read by the processors involved. Let the time taken for each polling round be t_{poll}, which is a linear function of m. It is necessary that all the processors in the same node be tightly synchronized, which can be accomplished via hardware clock synchronization similar to the one in [12]. After a processor gains control of the bus it sets the bus busy line high.

A polling round is necessary before every message is serviced. It will be shown later that the overhead caused by polling is insignificant. Recall that the tasks executing in the various processors at a node are assumed to be periodic. Let P_{max} denote the maximum period of all the periodic tasks[4] resident at that node. The deadline field will have to be large enough to represent this, since this is the largest possible deadline, though actual message deadlines are usually much smaller than P_{max}.

The resolution of the deadline field has to be determined. If all system time is counted in number of clock cycles, then the minimum resolution necessary is t_{poll} cycles (the time taken for the polling round in clock cycles). A finer resolution than this will not serve any purpose, since it takes at least t_{poll} cycles for the processor to get the bus. At least one bit of the poll number should change every t_{poll} cycles. For example, in the Fault-Tolerant Multiprocessor [11], the polling round takes 9 bit cycles. In this case, the LSB of the poll number's deadline field would represent the time period of 9 bit cycles or more.

The maximum number of bits needed in the deadline field is given by $log_2\ (P_{max}/t_{poll})$. The polling time is a linear function of the number of poll bit times in the polling round, which is the same as the number of bits m in the poll number. Thus $t_{poll} = cm$, where c is some constant. The number of bits (m) in the poll number is given by

$$\begin{aligned} m &= log_2\ N_{proc} + log_2\ N_{tasks} + log_2\ (P_{max}/t_{poll}) \\ &= log_2(N_{proc} N_{tasks} P_{max}/cm). \end{aligned} \quad (1)$$

The above expression reduces to the form $m = K2^m$, which can then be solved to obtain the number of bits in the poll number.

As an example, again, consider the Fault-tolerant Multiprocessor FTMP [11]. It has a clock rate of 8 MHz (125 ns). The real-time task workload on FTMP consists of three task classes of periods 40 ms (25 Hz), 80 ms (12.5 Hz) and 320 ms (3.125 Hz). The poll bit time for FTMP is 1 microsecond. If we use Eq. (2.1) to compute the number of bits m in the poll number, we have $t_{poll} = 15$ microseconds and $P_{max} = 320$ ms. An approximate computation reveals that we need 16 bits in the deadline field.

The number obtained by using the above expression for m is extremely conservative in nature. The design of a poll number with a 3 bit deadline field, 3 bit priority field, and 3 bit uniqueness field is shown in Fig. 6. The resolution in this case is the minimum task period, which is 40 ms.

2.3 Estimation of Message Deadlines

Estimation of the deadline for a message generated by a task is essential for determining the deadline field of the poll number. We propose a method to determine the deadline for a message generated

[3] Other encodings, which might result in a better performance, are possible.

[4] If some of the tasks are aperiodic, this would represent the maximum inter-actuation delay.

d2	d1	d0	p2	p1	p0	u2	u1	u0

DEADLINE (ms)	CODING			d2	d1	d0
0 - 20	0	0	0	1	1	1
20 - 40	0	0	1	1	1	0
40 - 60	0	1	0	1	0	1
60 - 80	0	1	1	1	0	0
80 - 100	1	0	0	0	1	1
100-200	1	0	1	0	1	0
200-300	1	1	0	0	0	1
300--	1	1	1	0	0	0

Figure 6: Possible design of a poll number.

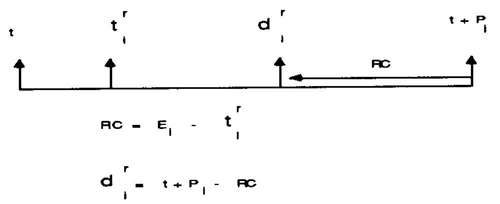


Figure 7: Estimating the deadline of a message.

by a real-tme task. It is assumed that all the tasks are initiated at the beginning of their periods. Further, each task executes on a single processor.

The period of a task i is depicted in Fig. 7. Since task i has been initiated at time t, the next initiation of the task will be at $t + P_i$, where P_i is the period of task i. During the course of its execution the task generates messages. The task then blocks at least until the message is put on the bus. It may or may not wait for the reply, before it resumes execution, depending on whether it executed a blocking send or a non-blocking send. The estimated time for completion of execution of task i (not including the delay while waiting for the bus) is given by E_i, which includes the time spent in blocking while waiting for a reply. The r-th message of this task is generated at t_i^r, when it has completed c_i^r units of execution time. In addition to the local clock, each processor keeps track (using a simple counter) of the execution time of its task which does not include the time spent by the task while waiting for the bus. At t_i^r, the task has generated its r-th message, and still needs the *residual computation time* $RC = E_i - c_i^r$ to complete execution. The deadline for the r-th message is $d_i^r = (t + P_i) - RC$, as the task has to send its message by d_i^r, if it is to complete the task by $t + P_i$.

E_i and P_i are known in advance and as soon as the processor starts executing an invocation of the task at time t, $t + P_i$ can be determined, and at any instant of time RC can be determined by subtracting the reading of the execution time from E_i. The value of deadline computed for the r-th message (d_i^r) is then complemented[5], resolved into coarser units, and loaded into the deadline field of the poll number. The priority and the uniqueness fields may be loaded at the same time.

2.4 Benefits of the Poll Number Approach

The bus access mechanism with the poll number makes the scheduling of messages flexible. A number of schemes can be implemented by varying the order and significance of the various fields making up the poll number. For example, we can implement closest deadline scheduling by loading the task deadline into the priority field. Priority driven scheduling can be implemented by making the priority field the most significant field. Likewise, deadline driven scheduling may be implemented by making the deadline field the most significant field.

We shall discuss the modifications needed to make the poll number based access mechanism compatible with the closest deadline scheduling scheme. The deadline field, as before, represents the message deadline. The priority field has been modified to hold the particular task invocation's deadline[6]. Let $TD_i = t_i + P_i$ and $TD_j = t_j + P_j$ denote the deadlines of task i and task j, respectively[7]. Let the uniqueness field of the poll number for task i be U_i, and that for task j be U_j. (By definition of the uniqueness field of the poll number, $U_i \neq U_j \; \forall i \neq j$.) With the above design, if a message from a task i, with a deadline d_i, competed for the bus with a message from another task j, with a deadline d_j, then task i would get the bus if and only if

1. $d_i \leq d_j$ or
2. $d_i = d_j$ and $TD_i \leq TD_j$ or
3. $d_i = d_j$ and $TD_i = TD_j$ and $U_i < U_j$.

It is easy to see that the above scheme is compatible with closest deadline scheduling which will schedule the message with the closest deadline, and in case of a tie will give preference to the task with the closer deadline.

It is also possible to make the polled bus access mechanism compatible with a priority driven scheme, where the various tasks have dynamically assigned priorities. In this scheme, the priorities of the tasks are assigned so as to satisfy some criterion, such as enabling a task to dispatch its message before its deadline. The proposed scheme is outlined in Fig. 8. When a task needs to use the bus, it generates a poll number based upon its priority at the moment, and the deadline of its message. The poll number is designed such that the priority field is more significant than the deadline field. Based upon this poll number, the task estimates the probability of the message missing its deadline. The probability of a message missing its deadline may be computed using Eqs. (3.4) and (3.5) in Section 3. In order to accomplish this, the task has to be aware of the service rates and the message generation rates of the other tasks at the node. A task has to be updated whenever the configuration of the node is changed.

[5]Since the processor with the highest poll number wins the polling round.
[6]If task i had been invoked at time t_i, then this field would hold $t_i + P_i$.
[7]These numbers are loaded into the priority fields of their respective poll numbers.

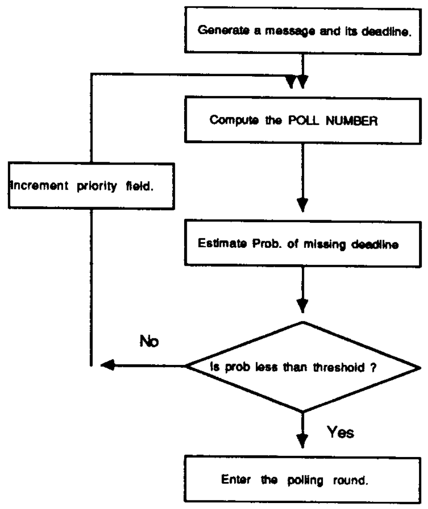

Generate a message and its deadline.
Compute the POLL NUMBER
Increment priority field.
Estimate Prob. of missing deadline
No
Is prob less than threshold ?
Yes
Enter the polling round.

Figure 8: Dynamic priority–based scheme using the poll number.

After generating a message, the task computes a poll number, and estimates the probability of the message missing its deadline. If the probability is higher than a predetermined threshold, the task recomputes its poll number after incrementing the priority field. The priority field is incremented until the probability of the message missing its deadline falls below the threshold. As soon as the probability falls below a threshold, the task then enters the polling round for the bus.

If all the tasks at the node increase their priorities to minimize their probabilities of missing deadlines, then it is highly likely that all tasks might end up having the same probability. In this case the scheme reduces to deadline driven scheduling. If the priority fields of the poll number are all equal, then the deadline fields will determine the task which will get the bus, as we have made the priority field the most significant field. This situation may be avoided by allowing only selected tasks to modify their priorities.

3 Performance Analysis

We have proposed an algorithm for bus access in the previous section. We shall compare the performance of the token bus with that of our polled bus. As stated before, our main objective is to minimize the number of messages missing their deadlines, and to honor a request as quickly as possible. The probability of missing a deadline will be computed for each processor in the node.

3.1 The Token Bus

A token bus is a single bus to which many processors are connected, and its access protocol is very similar to that of a token ring. A token is passed between processors on the bus and any processor which desires to use the bus has to capture the token. After using the bus, the processor releases the token to the logically adjacent processor on the bus.

We shall assume that only a single token is allowed to exist on the bus at any time if the bus is not being used by any processor. The token is regenerated when all the bits of the packet sent out by the processor is received by the processor again. This is possible as the data on the bus is accessible to all the processors on the token bus, which is an example of a broadcast bus. This acts as an acknowledgement for the sending processor. We shall assume a bit serial bus with a typical bit rate of 10 MB/s. There is a bus busy line which the processors can sample in order to check whether the bus is in use. We also assume that only a single task executes on each processor.

The three main events occurring in sending a message are:

1. The task generates a message.
2. The processor waits to get control of the bus.
3. The message is transmitted via the bus.

The generation of a message by a task executing on a processor is independent of the bus access mechanism used. We shall assume that the task generates a message (which has a deadline), waits for the bus and resumes execution only after the message is transmitted. The inter-message generation time is assumed to be exponentially distributed random variable with a mean time depending on the period of the task invocation. Thus, tasks with different periods have different mean inter-message generation times.

The time and mechanism involved in putting a message on the bus is also assumed to be independent of the bus access mechanisms used. The travel time of a message from one node to another depends on the bit rate of the bus and the physical distance separating these two nodes. We shall not analyze this since it is not affected by the bus access mechanism and hence does not help in contrasting the performance of the token bus with that of the polled bus. For similar reasons, the first event, generation of the message by the task, will not be analyzed further.

The time required to service a request is independent of the bus access mechanism used. It is assumed to be a random variable, which is identically distributed for all the processors in the node. The service time distribution directly affects the number of deadlines missed at a node. Hence any assumptions about this distribution shall be put off until all the parameters are analyzed. For the time being it suffices to say service time is identically distributed for all the processors of a node.

The chief parameter that we are interested in is the *bus access time*. The time taken to access the bus directly determines the number of deadlines missed due to unavailability of the bus. For a task i, the bus access time or wait time is denoted by W_i. We shall derive an expression for the wait time for the message generated by a task (processor).

The events occurring at a single node are depicted in Fig. 9. We are analyzing the system from the viewpoint of a single processor. At time $t = 0$, the processor had last successful possession of the bus. The processor may have used the bus or just passed the token along in a token bus case. The random variable T_c or the cycle time is defined as the next time the processor gets control of the bus. This cycle time depends on whether or not the other processors in the node use the bus during the cycle time.

A message is generated by task i at time t_i^r. Let S_i be the time

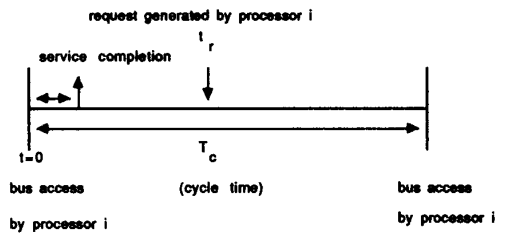

request generated by processor i
t_r
service completion
t=0
T_c
bus access by processor i
(cycle time)
bus access by processor i

Figure 9: The cycle time at a processor in the node.

required to service the message[8]. The service time distribution is assumed to be identical for all the tasks of a node. Let M be the number of tasks at at the node under consideration —which is the same as that of processors at the node— and let λ_i be the rate at which messages are generated by task i.

If there are no requests generated by any of the tasks on the token bus, the *cycle time*[9] is given by $T_c = T_{ring} + T_{token}$, where T_{ring} is the time taken for the token to travel on the bus once around the logical ring of processors and T_{token} is the time taken for a processor to recognize the token and put it back on the bus if it does not need to use the bus. The maximum value of T_c possible occurs when all the tasks at a node have messages to send. Thus, the value of T_c ranges from $T_{ring} + T_{token}$ to $T_{token} + T_{ring} + \sum_{i=1}^{M} S_i$.

First, we shall determine the cycle time T_c in terms of the service times. To compare the performance of the polled bus with that of the token bus, the cycle time T_c perceived by all the nodes is assumed to be the same, since a mean cycle time as opposed to an instantaneous cycle time at each processor is considered. Assuming that $T_c < P_i$, the probability that task i will generate a message during T_c is given by $\int_0^{T_c} \lambda_i e^{-\lambda_i t} dt$.

The cycle time T_c can be expressed as

$$\begin{aligned} T_c &= (\sum_{i=1}^{M} S_i \int_0^{T_c} \lambda_i e^{-\lambda_i t} dt) \ + T_{ring} + T_{token} \\ &= \int_0^{T_c} (\sum_{i=0}^{M} S_i \lambda_i e^{-\lambda_i t}) dt \ + T_{ring} + T_{token}. \end{aligned} \quad (2)$$

The above equation can be simplified to the following expression:

$$\sum_{i=1}^{M} S_i e^{-\lambda_i T_c} + T_c = \sum_{i=1}^{M} S_i + C, \quad (3)$$

where C is some constant. The above expression can be solved for T_c if the service time distributions are known.

The probability of a message generated by a task i missing its deadline can now be ascertained. Let d_i^r be the deadline of the current message generated by task i at time t_i^r. Denote the relative deadline $(d_i^r - t_i^r)$ of the current message of task i by D_i. The events occurring in a token cycle are shown in Fig. 10. The processor last had access to the bus at $t = 0$. The next access will occur at $t = T_c$, where T_c is the cycle time computed above. A message is generated by task i at time $t \leq T_c$, whose relative deadline is D_i.

[8]This is the time taken to deliver the message to the destination, once the bus has been acquired by the sending task.

[9]The time taken for the token to reappear at a processor after traversing all the other processors in a predetermined fashion. It is to be noted that we are talking of a mean cycle time here.

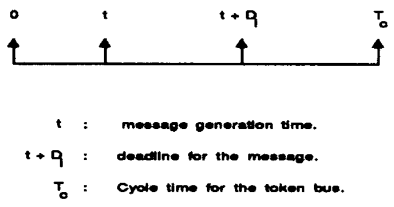

0
t
$t + D_i$
T_c
t : message generation time.
$t + D_i$: deadline for the message.
T_c : Cycle time for the token bus.

Figure 10: Events occurring during a token cycle.

The current message of task i will miss its deadline if and only if $t + D_i \leq T_c$. Because of the memoryless property of the inter-message generation times, the probability of a message being generated by a task in time t is given by $\lambda_i e^{-\lambda_i t}$. The probability of the current message of task i missing its deadline can be expressed as

$$P_{md}^{i} = \sum_{t=0}^{T_c} \lambda_i e^{-\lambda_i t} P(D_i \leq T_c - t). \quad (4)$$

3.2 The Polled Bus

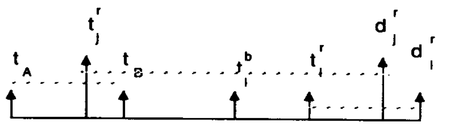
In this section we shall analyze the performance of a polled bus, access to which is determined based on the poll numbers generated by the various competing processors. In particular, we shall analyze the probability of a message missing its deadline. We assume that each processor in a node has only one task executing on it. This task is characterized by its repetition period and its user assigned priority which shall comprise the second field of the poll number.

In order to facilitate analysis, we shall define the following notation[10].

t_i^r :	time at which the current message of task i was generated.
t_j^r :	time at which the current message of task j was generated.
d_i^r :	deadline for the current message of task i.
d_j^r :	deadline for the current message of task j.
t_i^b :	time at which the bus was last acquired by task i.

The most recently generated message of a task whose deadline has not yet expired is termed as the *current message* of the task. After the expiration of the deadline the message loses its meaning in the case of real-time tasks. The various events which are significant in the analysis are depicted in Fig. 11. At t_i^b the current message of task i had the highest poll number, i.e., the closest deadline. In the absence of any competing messages from other tasks, the current message of task i generated at t_i^r will be honored. If there are other competing current messages from other tasks which have closer deadlines (larger poll numbers) than the current message of task i, then the current

[10]Any notation found in this section which has not been defined here will be the same as the notation used in the previous sections.



$$t_A = t_i^r - (d_j^r - t_j^r)$$

$$t_B = d_i^r - (d_j^r - t_j^r)$$

Figure 11: Event times in the case of polled bus.

message of task i will have to wait for the bus until all the other messages with closer deadlines have been honored. The time by task i in waiting for the bus is termed as the *bus access time* or *wait time* and is denoted by W_i.

We shall derive an expression for the mean wait time experienced by task i. The deadlines for the messages generated by the various tasks are determined by the method outlined in Section 2.3. In Fig. 11 we consider the current message of task j, generated at t_j^r, which is in competition with the current message of task i for the bus. The current message of task j shall be given the bus in preference to the current message of task i if and only if the deadline for the current message of task j (d_j^r) is less than the deadline for the current message of task i (d_i^r). It is now clear that the current message of task i will have to wait for the current messages of other tasks j for which $d_j^r \leq d_i^r$. Strictly speaking, the above condition should be the poll number computed by task j and must be less than the poll number computed by task i. Since we have assumed the deadline field to be the most significant field, in most cases [11] the condition on the poll numbers is reduced to the requirements on the deadlines.

From Fig. 11, it is clear that any competing messages from other tasks j must have been generated during the time span t_A to t_B in order to gain priority over the current message of task i. The probability of the above event is easily determined as the tasks are assumed to generate messages in a memoryless fashion. Therefore, the expression for the wait time W_i for the current message generated by task i is given by

$$\begin{aligned} W_i &= \sum_{j \text{ s.t. } d_j^r \leq d_i^r} S_j\left(\int_0^{(d_i^r - t_i^r)} \lambda_j e^{-\lambda_j t} dt\right) \\ &= \sum_{j \text{ s.t. } d_j^r \leq d_i^r} S_j(1 - e^{-\lambda_j (d_i^r - t_i^r)}). \end{aligned} \tag{5}$$

The probability of the current message of task i missing its deadline can now be determined. Let the relative deadline of the current message of task i be D_i where $D_i = d_i^r - t_i^r$. Therefore the probability of the current message of task i missing its deadline is given by

$$P_{md}^i = P(D_i \leq W_i). \tag{6}$$

The above expression may be used to arrive at the probability of a message missing its deadline. Performances of the token bus and the polled bus were simulated, and the probability of missing deadlines using the token bus was compared with that using the polled bus. It was assumed that all tasks had the same message generation rate and service times. Then the effect of varying the number of tasks, message service times and task execution times, for both the polled bus and the token bus, was observed. The results are shown in Figs. 12–14. In these simulations the deadline field of the poll numbers were assumed to have the actual deadline, not some quantized version of the deadline. It is expected that the performance measures will be less optimistic for the finite resolution case. Hence, these simulations represent the maximum possible performance achievable by using the poll number based bus access scheme (with the deadline field being the most significant).

[11] This is not true when the deadlines are equal.

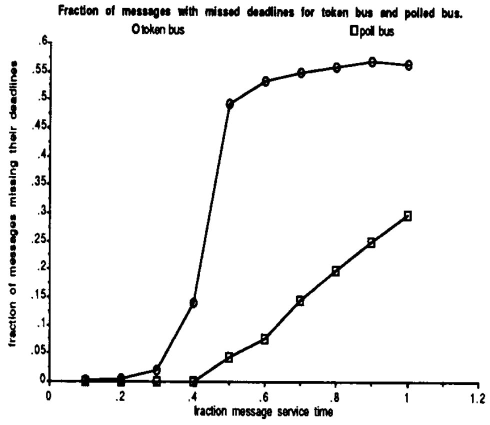


Figure 12: Effect of message service time on the fraction of messages missing deadlines.

The fraction of messages missing their deadline for various message service times (the time taken for the message to travel to the destination on the bus) is shown in Fig. 12.

The number of tasks was fixed at 8, the tasks all had the same period (40 ms), and the same execution time of 10 ms. The fraction of messages missing their deadlines increased with increasing message service times for both the token and polled buses. As expected, the polled bus had a lower fraction of messages missing their deadlines for all message service times.

In Fig. 13, the fraction of messages missing for different node configurations (number of tasks at the node) is shown. The fraction of messages missing their deadlines increased with increase in number of tasks for both token and polled bus. As expected, the polled bus had a lower fraction of messages, which missed their deadline, for all node configurations.

The fraction of messages missing their deadlines for different execution times of the tasks is shown in Fig. 14. The message service time, and the message generation rate were assumed to be the same for all the tasks, and were fixed. There were 8 tasks at the node. As before, all tasks were periodic with a period of 40 ms. The fraction

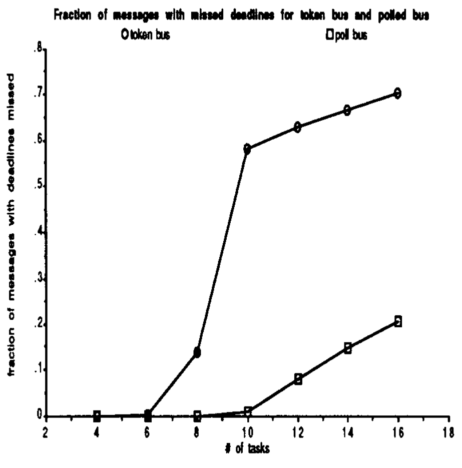


Figure 13: Effect of number of tasks on the fraction of messages missing deadlines.

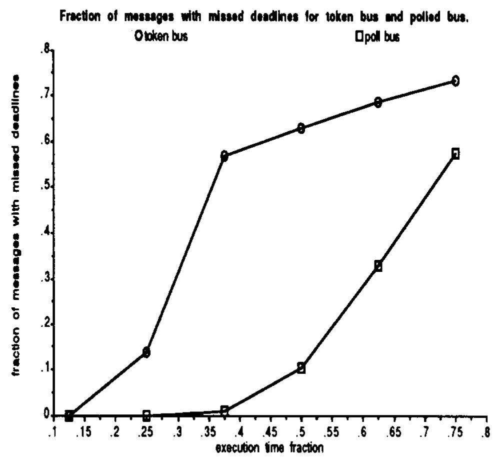


Figure 14: Effect of execution time on the fraction of messages missing deadlines.

of messages missing their deadlines increased with increase in the execution time of the tasks. For all task execution times, the polled bus had a lower fraction of messages missing their deadlines.

4 Conclusion

In any real-time system, the probability of a task missing its deadline should be kept as small as possible. Tasks often communicate with each other in a distributed real-time system in order to collectively perform some useful function. It is necessary that messages sent by various tasks meet their deadlines in order for the task to meet its deadline. We have proposed a polled bus access mechanism, using the concept of a poll number, to minimize the probability of messages missing their deadlines. The flexibility, decentralization and the performance improvement offered by the poll number approach make it particularly attractive for real-time distributed systems.

Acknowledgement

The authors are grateful to P. Ramanathan and B. J. Monaghan fc their assistance in preparing the final version of this paper.

References

[1] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real-time computer controllers and its application", *IEEE Trans. on Automatic Control*, vol. AC-30, no. 4, pp. 357-366, April 1987.

[2] R. H. Douglas, "IEEE token bus LAN implementation considerations", *Proc. IEEE COMPCON*, pp. 258-260, 1984.

[3] D. W. Jacobson, "High performance reliable token bus for the MAP network architecture", *Proc. IEEE Conf. on Local Computer Networks*, pp. 26-33, 1986.

[4] W. M. Gentleman, "Message passing between sequential processes: the reply primitive and the administrator concept", *Software Practice and Experience*, vol. 11, pp. 435-466, 1981.

[5] *Manufacturing Automation Protocol (MAP) specification (Draft)*, February 1986.

[6] J. T. Quatse, "An architecture for real-time cell control", *Control Engineering*, May 1987.

[7] ANSI/IEEE, *IEEE Standards for Local Area Networks: Token-Passing Bus Access Method and Physical Layer Specifications, ANSI/IEEE Std 802.4-1985*, Institute of Electrical and Electronics Engineers, Inc., 1985.

[8] J. F. Kurose, M. Schwartz, and Y. Yemini, "Multiple-Access Protocols and Time-Constrained Communication", *ACM Computing Surveys*, vol. 16, no. 1, pp. 43-70, March 1984.

[9] G. L. Choudhury and S. S. Rappaport, "Priority Access Schemes using CSMA-CD", *IEEE Transactions on Communications*, vol. COM-33, no. 7, pp. 620-626, July 1985.

[10] I. Chlamtac, A. Ganz, and Z. Koren, "Prioritized Demand Assignment Protocols and their Evaluation", *IEEE Transactions on Communications*, vol. 36, no. 2, pp. 133-143, February 1988.

[11] T. B. Smith and J. H. Lala, "Development and measurement of Fault-Tolerant Multiprocessor (FTMP)", *NASA Report*, vol. 1, , May 1985.

[12] K. G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious failure", *IEEE Trans. on Computers*, vol. C-36, no. 1, pp. 2-12, January 1987.

Describing Real Time Systems Using PPA and XYZ/E

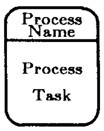
Jianbai Wang **Murat M. Tanik**

Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275

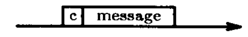
ABSTRACT

In current software development environments, formal approaches provide precise representation of design for correct program generation while informal techniques facilitate software developers with flexibility and friendly interfaces. They are frequently combined to achieve effective supporting for software development. Among the approaches, PPA, a data-flow diagram system enhanced with process port concept, and XYZ/E, a temporal logic based language system[2], are proposed to be used in real-time system design. By describing a cruise control system[1], a commonly shared example, using PPA and XYZ/E, this paper investigates their capabilities in describing real time systems.

Process Port Analysis (PPA) is a diagram system for information system design. It expresses a system as a graph with nodes representing processes or other kinds of entities of the system, and edges among nodes representing control or communication. Process nodes and information flow are represented by



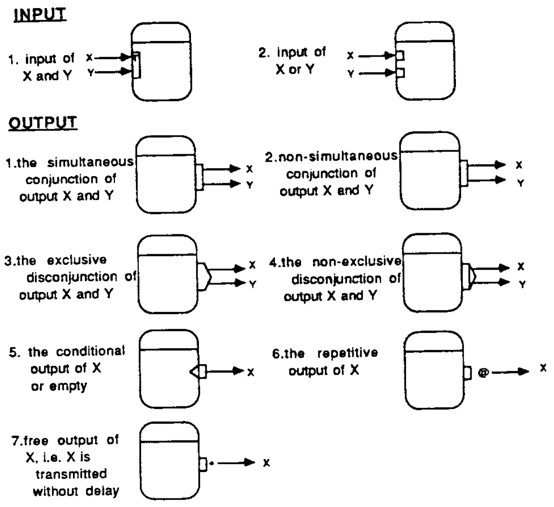
and



The "c" is the channel name of the data-flow. Without any information attached, the flow is a control signal. The following figure shows refinement of the process nodes with input/output analysis.

When a control flow is directed to a process as its input it is used to trigger the process. The letter "T" in a process box is used as a trigger mark. A control flow comes from a process as its output has the function to terminate the process. Conditional and repetitive output could not be control flow. In conjunctive outputs, there must be at least one flow for control in each case, while all are control in disjunctive output.

The PPA description of the CCS is given in Figure 1. In the diagram, All processes besides A5 are triggered by the system-on/off signal meaning that these four processes will keep on working until the system switch is turned off since there are control flows directed to themselves. One of the output from Process A4 is a control flow, and is conditional, to trigger Process A5. In Figure 1, the sequence of program execution and the concurrency among processes are reflected by


INPUT
1. input of X and Y
X
Y
2. input of X or Y
X
Y
OUTPUT
1.the simultaneous conjunction of output X and Y
X
Y
2.non-simultaneous conjunction of output X and Y
X
Y
3.the exclusive disconjunction of output X and Y
X
Y
4.the non-exclusive disconjunction of output X and Y
X
Y
5. the conditional output of X or empty
X
6.the repetitive output of X
X
7.free output of X, i.e. X is transmitted without delay
X

The Input / Output Analysis of Process

different combinations of control output. The control flow and data flow can be distinguished even if they share the same channel. Given such representation, one can clearly deduce the data communications characteristics and control dependencies among processes and other entities.

XYZ/E, as stated by its developer , "is a linear time temporal logic system and is also a real programming language, ... It can be used to describe systems from abstract level to effectively executable level"[2]. In XYZ/E, a program is a sequence of Conditional Elements (CEs) which have the following form :

$$\Box\,[\,A1;\ A2;\ \dots\ An\,]$$

where Ai, i = 1, .., n are CEs. Each CE is a well formed formula (wff) of

$$\#lb = y \;\&\; P \Rightarrow @\ (\,Q \;\&\; o\#lb = z\,)$$

which represents a state transition. @ is either o (next time), or $<>$ (eventuality). A name with a leading sharp sign "#" is a temporal variable. Each CE denotes that if it is now at the state that #lb = y and P true then it implies ("$\Rightarrow$") next time it will be at state that #lb = z and Q true. In a more conventional sense, it could be understood as that y and z are two program labels before and after the state transition, P and Q are conditional and statement respectively. Defined by XYZ/E as a control variable, #lb always refers to the current label of a program. XYZ/E also uses temporal operators $\Box$ (necessity) and the corresponding operators for past time of o, $<>$, and $\Box$. Statement in XYZ/E is an equation o#v = exp which assigns the value of expression exp to the variable v. In case of concurrency, each process should have its own control variable.

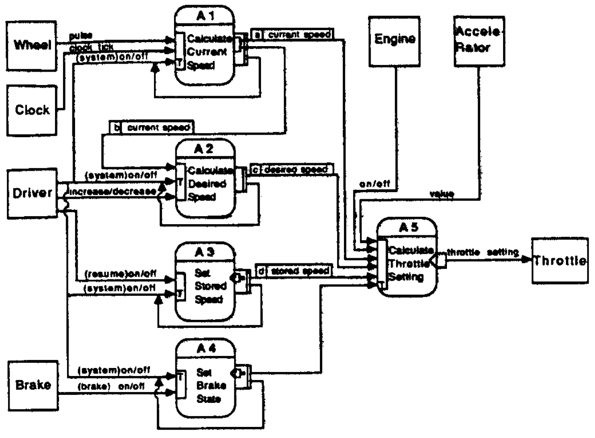


Figure 1. The Cruise-Control System in PPA

As an example, Process A4 and A5 of the CCS represented in PPA is also represented in XYZ/E as follows:

```
A4 { Set Brake State }
 ( 1b4 = start4   ⇛ o 1b4 = A4 ;
   1b4 = A4 & ( Brake = on )
                 ⇛ ◇ ( o 1b4 = A 4)
   1b4 = A4 & ( Brake = off )
                 ⇛  ◇ ( o 1b4 = A 4 )
                  & ◇ ( o 1b5 = A5 ) )

A5 { Calculate Throttle Setting }
 ( 1b5 = A5  & ( Engine = on )
                 ⇛   ( CTS )
                  & ◇ ( SEND throttle_setting TO THE THROTTLE )
                   & o1b5 = STOP ;
 ( 1b5 = A5  & ( Engine = off )
                 ⇛  o 1b5 = STOP )
```

From the XYZ/E description of Process A4, it is clear that "Brake on" is the exact condition for triggering A5. Process A4 will continuously perform its execution until the system is off while Process A5 will stop as a result of executing formula "olb5 = STOP" when its task is finished. The temporal operator $<>$ (eventually) indicates data and/or control output. The assignment without the operator indicates state transition within a process. XYZ/E provides better precision at the same description level: the time dependent synchronization actions of the processes such as, start, stop, and continuous execution.

A scenario of the CCS demonstrates that the XYZ/E description does capture the system function correctly [4]. It is also shown that the descriptive advantages of PPA, other than intuitive graphical notation, exist in XYZ/E, as well. From the formal description of XYZ/E, executable codes can be more readily derived. Further, both of the approaches (PPA and XYZ/E) provide capability for task decomposition. In addition, a verification theory of XYZ/E exists[2,3].

Acknowledgement

We are grateful to Prof. C. S. Tang for discussing his work on XYZ/E system in his visit of our department and his subsequent critique of this presentation.

References

[1] Booch, G., "Object-Oriented Development" IEEE Transaction On Software Engineering, Vol. SE-12, No. 2, Feb. 1986

[2] Tang, C. S., "To Unify Programming With A Temporal Logic Language System" Working paper, Carnegie-Mellon University, 1987

[3] Tang, C. S., Personal Communication. (C. S. Tang, Professor, Division Five, Institute of Software, Academia Sinica, P.O.Box 8718, Beijing, P.R. China)

[4] Jianbai Wang, Murat M. Tanik, "Describing Real-Time Systems using PPA and XYZ/E", TR-88-CSE-10, SMU, 1988