Jitter Reduction in a Real-Time Message Transmission System Using Genetic Algorithms

Jorge Barreiros[1,2] jmsousa@isec.pt,
Ernesto Costa[2] ernesto@dei.uc.pt
José Fonseca[3] jaf@det.ua.pt
Fernanda Coutinho[1] fernanda.coutinho@isec.pt

Abstract- The wide use of fieldbus based distributed systems in embedded control applications triggered the research on the problem of transmission network induced jitter in control variables. In this paper we introduce a variant of the classical Genetic Algorithm, which we call Progressive Genetic Algorithm, and show how it can be used to reduce jitter suffered by periodic messages. The approach can be applied either in centrally controlled fieldbuses or in synchronized ones. The algorithm was tested with two well-known and widely used benchmarks: the PSA, coming from automotive industries and the SAE from Automatic Guided Vehicles. It is shown that it is possible to eliminate completely jitter if the adequate transmission rate is available and, if not, a satisfactory reduced jitter can be obtained.

1 Introduction

Distributed systems are today widely used in control applications. In these, control tasks such as data acquisition, control algorithm execution, system identification, actuation, are often carried on in different nodes of the system. While in stand alone classical controllers sampling period can usually be kept as close as possible to the desired value, in distributed systems this may not be true due to the need to transmit data over a shared communication medium.

Presently most distributed systems either industrial or embedded, rely in a serial communications infrastructure known as fieldbus [Thomesse 98] to interconnect a set of nodes. Some sort of arbitration is then used to decide, in each time instant, which information will be transmitted in the bus [Almeida et al 99]. When periodic variables such as control parameters are to be transmitted, it is usually possible to impose an adequate average transmission period. However, due to the interaction of the different variable periods and, often, of sporadic or aperiodic traffic, it is rather difficult to obtain constant time intervals between successive instances of the same periodic variable. This variation is called jitter and may be formally defined as: instantaneous jitter of instance k of periodic parameter i belonging to the set of messages, $j_{i,k}$, is:

$$j_{i,k}=ts_{i,k} - (k * T_i + \phi_i)$$

where $ts_{i,k}$ is the start time of the transmission of the instance, T_i is the period and ϕ_i is the initial phasing at the start-up of the system. Thus, the instantaneous jitter is the difference between the expected beginning of transmission $(k * T_i + \phi_i)$ and actual transmission start $ts_{i,k}$.

The overall system jitter (OSJ) can be defined as:

$$OSJ = \sum_i \sum_k j_{i,k}$$

The problem of jitter in periodic control variables has been identified recently and its consequences have been studied. [Hong 95] refers to several scenarios of this problem such as the case when more than one sample occurs in the same period which leads to data rejection and when there is no sample in that interval which is known vacant sampling. [Stothert and MacLeod 98] demonstrated the degradation of performance in feedback control loops subject to jitter in the sampled and in the actuation variables (what they called read-in and read-out jitter) due to their transmission in a distributed control system. In [Juanole 99] the effect of jitter due to message transmission in CAN – Controller Area Network [Bosch 91] is shown to affect the phase margin of a control loop. These and other examples demonstrate the need for further research in jitter minimization as it is pointed in [Decotignie 99].

In this work, we use genetic algorithms for reduction of transmission network induced jitter in control variables in a distributed embedded system. A variant algorithm, which we called Progressive Genetic Algorithm (proGA), is proposed and compared with the simple one. It is shown that proGA enhances the performance of the simple genetic algorithm for this specific problem. Moreover the variant algorithm is also well suited for application in a real-time online optimization device.

This paper is structured as follows: in the next section we introduce the genetic algorithms we used and present how we model the problem; in section 3 we describe the experiments we conduct using two benchmarks from the automotive industry; finally in section 4 we present some conclusions and discuss direction for future work.

[1] Superior Institute of Engineering of Coimbra, Quinta da Nora, 3030 Coimbra, Portugal
[2] Centre of Informatics and Systems of University of Coimbra, Polo II, Pinhal de Marrocos, 3030 Coimbra, Portugal
[3] Dep. Electronic and Telecomunications of University of Aveiro, Campus Santiago, 3800 Aveiro, Portugal

2 Progressive Genetic Algorithm

2.1 Genetic Algorithms

Genetic Algorithms (GA) are stochastic search procedures inspired by the biological principles of natural selection and genetics (Holland 75) (Back et al. 97). In spite of their possible variants, GA can be described by the following general procedure:

```
Procedure GA
  t=0;
  Intialize P(t);
  Evaluate P(t);
  While stoping_criterion_false  do
   t = t+1;
   P'(t) = select_from P(t-1);
   P''(t) = use_op_modification P'(t);
   Evaluate P''(t);
   P(t) =  merge P''(t) , P(t-1)
End_do
```

The GA starts with a set of candidate solutions called a *population*, usually defined randomly. Each element of that initial population, called an *individual*, is then evaluated using a *fitness function* that gives a measure of the quality of that element. Each individual is in fact an aggregate of smaller elements or units, which are called *genes*. Each gene can have different values or *alleles*. The algorithm enters then a cycle in order to generate a new population. It starts by probabilistically *selecting* the fittest individuals. Then they undergo a modification process, using genetic inspired operators like *crossover* or *mutation* that will eventually alter the alleles of some genes. Finally, the old and new populations are combined and the result becomes the next generation that will in turn be evaluated. The cycle stops when a certain condition is achieved (for instance, a pre-defined number of generations). The algorithm just described generally works with a low-level representation of each individual called its *genotype*. Nevertheless, in complex problems, the fitness function acts upon a high-level representation of an individual, its *phenotype*, making necessary to use *decoders* from genotypes to phenotypes.

The success of GA algorithms is linked to their ability to solve difficult problems: problems where the search space is large and multi-modal. This is the case of the problem described above where the goal is to minimize the overall message transmission jitter in a priority-based real-time communication system. But for a GA to be more efficient than traditional algorithms it is crucial to incorporate in it problem-specific knowledge. This is achieved by choosing a good data structure to represent the chromosome and also by "tuning" the genetic operators (Michalewicz 99).

2.2 The elements of a simple GA (sGA)

For this particular problem, the genotype will be a vector of integers where each integer represents the initial offset of one message. For a message M each integer will be within the range of $[0,P_M-1[$, where P_M is the period of that message. All the phases for which the previous condition holds true are said to be *feasible*.

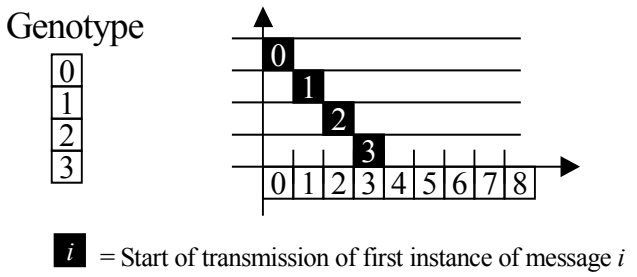

Figure 1 -Phases representation

Our crossover operator mimics the one that occurs in natural sexual reproduction. Crossover uses two individuals of the population (the *progenitors*) to generate two new individuals (the *descendants*). We use *one-point* crossover. In this case, a single random crossover point R is chosen, within the range of 0 and N (where N is equal to the number of messages). For the phases between 0 and R, the resulting descendent is equal to one of the progenitors. For phases between R+1 and N, it is equal to the other progenitor. A second descendent is created inverting the order of the progenitors. In this system, crossover probability is 1. This means that all the descendents are created with the crossover operator.

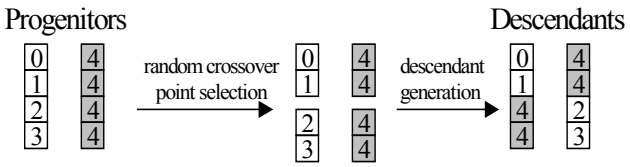

Figure 2 - One-point crossover

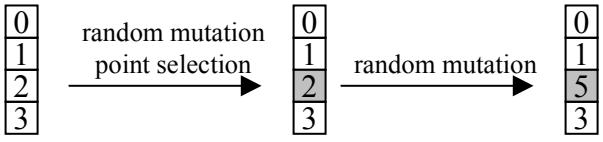

Figure 3 - Mutation operation

We used simple mutation (see Figure 3 above): we change probabilistically one individual's gene. The probability is 3%.

The fitness of each individual is evaluated by running a simulation of the system operation that calculates the OSJ. The OSJ is the sum of the partial jitter of each message. The fittest individuals are those for which this value is smaller. The simulation process will be described later.

The genetic algorithm stops when a predetermined number of consecutive descendants are not included in the

population. This happens when they are all worse than every individual already on the population.
The progenitors are selected proportionally to their fitness.
At each iterative step, one or two of the worst individuals of the old population may be replaced by one or both of the generated descendents. In this way, a new population is generated. This replacement takes place when the fitness of the later is better than the fitness of the former (steady state like GA mechanism). This will in principle improve the average fitness of the population.
The initial population is random generated but constrained to feasible individuals.

2.3 The Simulator

Three different simulation mechanisms were tested: a simple time-driven simulator, a hierarchical simulator and an event-driven simulator. The first used a direct approach to the problem, updating the system's state in every tick of the clock. This was the one with least performance of the three simulators.
The second one is a hierarchical simulator that addressed the problem by decomposing the system into simpler sub-systems. The simulator initially calculated the OSJ for a simpler system with only the two messages with highest priority, and then proceeded incrementally adding messages until the solution for the complete system was obtained. Although this solution scaled well and achieved more performance than the simple simulator (albeit an initial overhead noticeable on smaller systems), it's memory requirements grew very rapidly as the system complexity increased. The best performance was obtained using an event-driven simulator.
In the third approach, the simulator uses two integer vectors for holding information about the system's state. One vector holds the instants in which each message will be next ready to transmission. The other vector holds information on the elapsed transmission time of the messages currently being transmitted. Message information is ordered by priority: in position 0, the message with highest, in position 1, the second highest message, and so on.
Using information from the first vector it is possible to determine how long will it take before the next message is ready. The system's state held in the second vector can be updated accordingly.
The simulation stops when an overall system period has been simulated. This period is equal to the LCF (least common factor) of the periods of the messages.
The picture bellow shows the simulator's output for a message set with periods {4,6,8,12} and transmission times of {1,2,1,2}. The #'s show actual message transmission and the ~'s represent the jitter. The time axis' offset is caused by the transient state removal, as explained ahead.

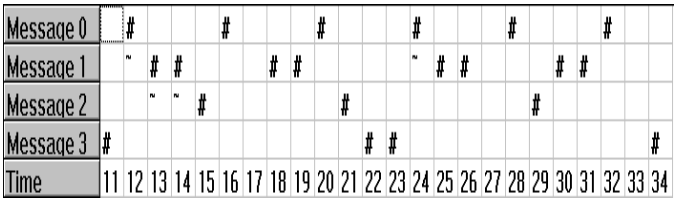

Figure 4 - System simulation example

The following algorithm is a simplified version of the simulator's algorithm, not including transient analysis and statistic data gathering.

A=vector with current transmission times of messages
B=vector with time remaining befores each message is again ready for transmission
CM=index of current message being transmitted (-1 = none)
Set A to 0
Set B to initial system phases
Set CM= -1
While (not(StoppingCondition))
Determine time interval before next message is transmitted (TT=min(B))
Update A:
    While(TT>0)
    Set CM to the message being transmitted now
    (if CM!=-1 it's the index for the first non-zero value of vector A, else it is CM)
    (if CM=-1 and A hasn't got any non-zero values, goto Update B:)
    MTM is the remaining transmission time of the message that is currently being transmitted (CM)
    If (MTM-TT<=0), current message has been transmitted:
    Update A so that message CM's remaing transmission time is 0
    Set CM= -1
    Else
    Update A so that message CM's remaing transmission time is MTM-TT
    TT=TT-MTM
    Wend
Update B:
    TT=min(B);
    Subtract TT from every time in B.
    For each value in B=0, set it to the corresponding message's period
Wend

Normalization

A normalization process was used to ensure that the phase of the message of greater priority would always be zero. Because all the phases are relative to the same instant, the crossover operator is more efficient. Additionally, the search space is reduced in one dimension.

Transient Analysis

There is an initial transient state in the simulation, which is caused by the absence of transmission channel use at instant

0. This effect is noticeable because there may be messages still transmitting when the simulation ends. These messages "wrap around" the overall system period and can't be determined when system simulation begins. Using an initial transient period in the simulation, in which no statistical data is gathered, solves this problem. This transient period should be long enough so that the effects of the "cold" start are eliminated.

The duration of the transient state is determined as follows. If the simulation starts at instant 0, then the worst case happens when all the messages are made ready for transmission in instant –1 (except for the most priority message, which always begins at instant 0 because of the normalization process described above). The worst case duration of the transient state is determined by simulation of system execution in this worst case scenario, and it is equal to the time where the message with least priority ends transmission for the first time. From this moment forward, the system's state will no longer depend on initial conditions.

2.4 Progressive Genetic Algorithm (proGA)

Execution time of the simulation is dependent on the LCF of the message set. This fact has lead to the development of a variant of the standard genetic algorithm, which reduces the number of messages simultaneously analyzed. The variant, called Progressive Genetic Algorithm (proGA), increases the number of analyzed messages in an iterative way until a solution for the complete set is found. The overall operation of the algorithm is described bellow:

- First, the problem for the set of the two most priority messages is solved.
- Next, the problem for the set of three highest priority messages is solved using the solutions from the previous step,…
- …, and so on, until the problem for the full set of messages is solved.

Besides performance issues, an additional advantage of the proGA is the existence of a partial solution for the problem prior to completion of the algorithm. The initial solutions for a small number of messages can be used in an on-line optimization device that produces best-effort solutions to a communication scheduler. Since most of the bandwidth is used by the highest priority messages (considering a RMS[4] scheduling mechanism is used), these partial solutions could be, in most cases, quite near the real optimum solution.

The following changes were made to the original genetic algorithms.

- The sGA is used to solve the problem for the set of the two highest priority messages
- The solutions for the set of N-1 messages are used to initialise the population used to solve the problem with the N highest priority messages in the following way:
- Determine the set C of individuals that have maximum fitness for the set of N-1 messages
- Each individual of the initial population for N messages is built from a randomly chosen member of C. The phases of the N-1 highest priority messages will be equal to those of the chosen individual, while the phase for the Nth message will be a new, random, integer in the admissible range $[0, P_N[$.

The complete algorithm is presented below:

```
N =Vector with description of the two greatest priority messages
NP=2; //Progression counter
P=Random population of individuals for the two greatest priority messages
PopSize= Size of population (number of individuals)
Do
  // Solve the problem with the sGA for the message set N, using initial population P
  S=FinalPopulation( sGA(N,P) );
  //  Add one more message to the the problem input
  NP=NP+1;
  Add(N, Description of the NP^th biggest priority message)
  // Select a breeding pool that includes all the individuals in S that have maximum fitness
  BreedingPool=Max(S)
  P=[]; // Clear P
  // Build new population by crossing and mutating from breeding pool
  While (Size(P)<PopSize)
      I = ApplyGeneticOperators(BreedingPool);
      Append(I,random-integer-in[0,NP^th message period [)
      //Merge individuals
      Merge(P,I );
  Wend
While NP<=NumberOfMessages
```

3 Experiments

We conducted a series of experiments for assessing the performance of the proGA versus the sGA. Both algorithms were tuned similarly:

- Crossover probability: 100%
- Mutation Rate: 3%
- Population Size:200
- Stopping criterion: 200 iterations without improvement of worst individual of population

Some experiments were additionally restrained to having solutions with alleles multiple of a base value (10 μs). Higher base values originate solutions that can be implemented by timers with smaller resolutions.

[4] Rate Monotonic Scheduling – Scheduling policy where messages with greater priority are those with smaller periods. If RMS is used, then it is fairly simple to prove the schedulability of the system, so it is widely used in real time communication systems.

We assumed a Controller Area Network (CAN) fieldbus was being used, with transmission rates of 250 Kbit/s and 125 Kbit/s.

Testing Sets

Two benchmark sets were used as test input for the algorithms. These benchmarks were developed in the automotive industries with the purpose of modeling actual message transmission requirements on automobile vehicles (PSA - Peugeot Societé Automobile) or Automated Guided Vehicles (SAE - Society of Automotive Engineering). The benchmarks specify the system messages' length (in bytes), and period of transmission. These benchmarks are normally used for testing fieldbus architectures, network protocols, scheduling policies and related experimentation.

The PSA benchmark specifies 12 messages, with periods ranging from 10 ms to 100 ms. The messages' length varies widely, from 1 to 8 bytes. The SAE benchmark includes 53 messages with periods from 5 ms to 1 sec, with a fixed size of one byte (in a CAN network).

The message sets were manually scheduled with the RMS scheduling policy. A worst case situation in which all messages are released simultaneously in the beginning of system's operation was considered. This ensures that the set is still schedulable no mater what initial phasing is imposed to each of the messages.

Results: proGA vs. sGA

In the following tables, each experience is one independent run of the algorithm. Improvement is rounded, not truncated.

PSA – 125k Resolution : 1 µs	*sGA*	*ProGA*	*Improvement (1-(Res proGA/Res sGA))*
Number of experiences	20	20	
Best OSJ (µs)	12640	12640	0%
Average OSJ (µs)	18626	13947	25%
OSJ Variance	18609324	1431317	92%
Best execution time (s)	115	93	19%
Average execution time (s)	222	160	28%
Worst execution time (s)	337	308	9%
Execution time variance	4925	3038	38%

PSA – 125k Resolution : 10 µs	*SGA*	*proGA*	*Improvement (1-(Res proGA/Res sGA))*
Number of experiences	20	20	
Best OSJ (µs)	12728	12640	1%
Average OSJ (µs)	18529	13365	28%
OSJ Variance	18505900	554941	97%
Best execution time (s)	104	81	22%
Average execution time (s)	235	171	27%
Worst execution time (s)	392	311	21%
Execution time variance	6198	3694	40%

SAE – 250k Resolution : 1 µs	*SGA*	*proGA*	*Improvement (1-(Res proGA/Res sGA))*
Number of experiences	20	20	
Best OSJ (µs)	1497	0	100%
Average OSJ (µs)	5547	68	99%
OSJ Variance	17480887	5132	100%
Best execution time (s)	1171	298	75%
Average execution time (s)	1797	540	70%
Worst execution time (s)	2627	937	64%
Execution time variance	177948	31856	82%

SAE – 250k Resolution : 10 µs	*SGA*	*proGA*	*Improvement (1-(Res proGA/Res sGA)))*
Number of experiences	20	20	
Best OSJ (µs)	468	0	100%
Average OSJ (µs)	4663	178	96%
OSJ Variance	10684342	94820	99%
Best execution time (s)	1063	435	59%
Average execution time (s)	1892	739	61%
Worst execution time (s)	2712	1132	58%
Execution time variance	162816	36933	77%

Results Analysis

The best solution given by the proGA is always better than or equal to the solution from the simple GA. The average OSJ is largely improved in all the experiences, from 25% in the worst case (PSA-125k-1µs) up to the best improvement of 99% in (SAE-250k-1µs). There is also a very significant decrease in the variance of the results. These improvements may be explained by the following factors:

- The proGA progressively identifies zones of the search space that are more likely to have good solutions for the problem, and directs the search towards those zones. This results in increased time-efficiency of the algorithm.
- The result of the sGA is more dependent on the random initial population than the proGA's result. In fact, no matter what the initial population is, the proGA tends towards "good" zones of the search space soon enough.

Concerning the speed of the algorithm, the proGA is more efficient than the simple GA. Improvements range from 27% for the PSA benchmark, up to 70% in the SAE benchmark. Improvement in this benchmark is superior because it is more complex than the PSA, having a greater number of messages and bigger system period. The proGA scales better than the sGA, and thus offers increased improvement. Thus, it can be seen that the additional overhead introduced by the proGA is largely compensated by the superior convergence speed for an optimal solution.

The timer resolutions of 1 µs and 10 µs seem to offer reasonably similar optimal solutions. In fact, additional testing showed that resolution could be lowered to 100µs without significant impact on the quality of the solutions (in some cases, degradation only occurred at 1000µs resolutions).

Progressive Phase Optimization

Figure 5 below shows the evolution of the number of optimized phases during execution of the proGA. The algorithm was tuned as described above.

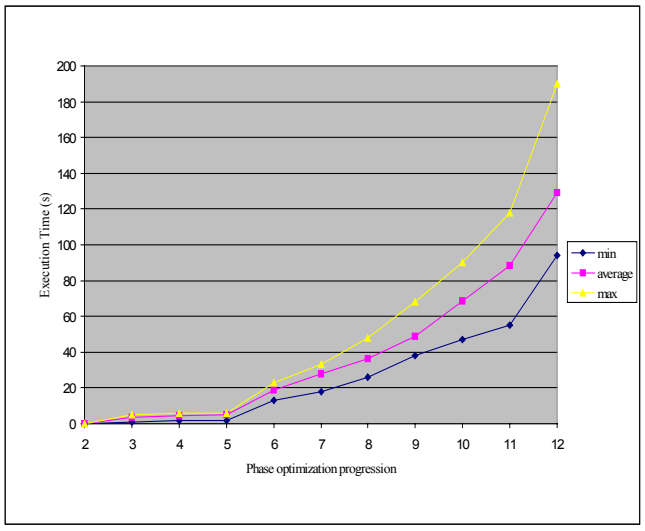

Figure 5 – Progressive phase optimization

It can be seen that most of the execution time is spent optimizing the least priority messages. These tend to have little impact on the OSJ, and so it is possible to develop an on-line optimization device that offers near-optimal best-effort solutions.

4 Conclusions and Future Work

In this paper we studied techniques for reduction of network induced jitter in message transmission, using genetic algorithms.

Two variant methods were analysed; a simple steady-state based GA and a new approach we called proGA (progressive GA). This new approach was developed to provide adequate performance and on-line optimisation capabilities. Additionally, we also studied the effects of variation of timer resolution on minimum overall system jitter.

The experimental results show that there is a significant increase in the quality of the solution when the proGA is used instead of the simple GA. The variant GA is also more efficient, offering considerable improvements on execution speed.

Although the technique imposes a relatively high computational overhead, at least considering the usual processors available in these type of systems, it is possible to apply it incrementally. The initial solutions for a small (5,6) number of messages are usually obtained rather quickly, making it feasible to develop an on-line optimization device that produces best-effort solutions to a communication scheduler. An adequate system or co-processor can then work on-line on messages phasing promoting a continuous reduction of jitter along system operation. This opens additional interesting possibilities for the on-line admission of new real-time messages.

We have also found out that the timer resolution can be decreased from maximum resolution without significant loss of quality of the solution. This means that on-line optimization can be made with simpler devices with lower frequencies.

Acknowledgments

The work of J. Barreiros and E. Costa was partially supported by the Portuguese ministery of Science and Technology under Program PRAXIS XXI.

Bibliography

[Back et al 97] Back, T., Fogel, D., Michalewicz, Z. (eds.) *Handbook of Evolutionary Computation*, Oxford University Press, New York,1997

[Almeida et al 99] Almeida, L., Chavez, M.L., Fonseca, J.A., Thomesse, J.-P.. "Real time communications in manufacturing" Proceedings ISAS'99 The 5th. Int'l

Conference on Information Systems Analysis and Synthesis, Orlando, USA, July/August.

[Bosch 91] "CAN specification version 2.0 - Technical Report", Bosch GmbH, Stuttgart, Germany, 1991.

[Decotignie 99] Decotignie, J.D., "Some Future Directions in Fieldbus Research and Development", Proceedings FeT '99 - Fieldbus Systems and Applications Conference, Magdeburg, Germany, September 1999.

[Holland 75] Holland, J. *Adaptation in natural and artificial systems*, University of Michigan Press, Michigan, 1975

[Hong 95] Hong, S., "Scheduling Algorithm of Data Sampling Times in the Integrated Communication and Control Systems", IEEE Transactions on Control Systems Technology, Vol. 3, Nº 2, June 1995.

[Juanole 99] Juanole, G. "Modélisation et Évaluation du. Protocole MAC du Réseau CAN", École d'été ETR'99 – Applications, Réseaux et Systèmes, ENSMA, Poitiers – Futuroscope, France, September 1999.

[Michalewicz 99] Michalewicz, Z., Genetic Algorithms + Data Structures = Evolution Programs (3rd, revised and extended edition), Springer-Verlag, Berlin,1999

[Navet and Song 97] Navet, N., Song, Y.-Q., "Performance and Fault Tolerance of Real-Time Applications Distributed over CAN (Controller Area Network)", CiA – CAN in Automation Research Award, 1997.

[Stothert and MacLeod 98] Stothert, etal "Effect of Timing Jitter on Distributed Computer Control System Performance", Proc. 15 IFAC Workshop DCCS'98 – Distributed Computer Control Systems, September 1998.

[Thomesse 98] Thomesse, J.P., "*A Review of the Fieldbuses*", *Annual Reviews in Control,* 22 pp. 35-45*,* 1998.

[Tindell and Burns 94] Tindell, K., Burns, A., "Guaranteeing Message Latencies on Control Area Network (CAN)", Proceedings of ICC'94 (1st International CAN Conference), Mainz, Germany, 1994.