SOFTWARE IMPLEMENTATION SIMULATION TO IMPROVE CONTROL LAWS DESIGN

Rémy KOCIK (speaker), Mongi BEN GAID, Rédha HAMOUCHE
COSI LAB, ESIEE, Cité Descartes - BP 99 - 2, Bd Blaise Pascal - 93162 Noisy-Le-Grand Cedex
Tel: +33 (0)1 45 92 67 86 - Fax: +33 (0)1 45 92 66 99
kocikr@esiee.fr, bengaidm@esiee.fr, hamouchr@esiee.fr

October 31, 2005

Symposium Topic

Electronic integration of sensors and actuators: Tool and Methodology specifications for safe embedded systems.

Keywords

Control, real-time implementation, distributed emmbedded systems, simulation, design lifecycle.

1 Introduction

The challenge in embedded systems is to design more and more complex applications while taking into account strong cost constraints such as financial costs, consumption and footprint. To reduce hardware cost, the automotive industry has chosen to use distributed architectures made of ECUs (Electronic Control Unit) linked together by networks such as CAN, FlexRay, ... [8]. Processors are placed near sensors and actuators, sensors data may be used by many functions : the number of sensors and the wiring may be reduced.

This increasing hardware complexity impact on software complexity design : computations have to be placed and scheduled on ECUs and communications must also be scheduled on the network taking into account the real-time constraints.

The design of a real time control embedded system usually starts by a modelling step where control engineers describe mathematically the behaviour of the system to be controlled (control law synthesis). The behaviour is validated by simulation, then the control laws are implemented on the hardware architecture by computer engineers.

The software on each ECU is written and executed on the architecture, then the system has to be validated. In most cases, the obtained behavior of the system may be quite different from the behavior simulated by control engineers.This difference is mainly due to the control laws real-time execution : computations and network communications introduce variable delays which are known to degrade control performances. To take them into account, control laws have to be tuned and the design cycle must be iterated until the behavior of the system meets the design objectives. These numerous iterations increase the design lifecycle time.

At the simulation step, the delays induced by real-time implementation are not known. They depend mainly on the scheduling algorithms, the priority choices, the communication protocols and the low level implementation mechanisms[3, 7]. Taking into account these delays in the simulation step may help to tune the control laws earlier and, by this way, may decrease the number of iterations[6].

In this paper, we describe how a real-time distributed implementation may impact on control performance. Then we show on a suspension car controller design example how introducing implementation model in the simulation may help engineers to improve the control law synthesis and may also help to make implementation and hardware architecture design choices.

2 Control embedded system design

2.1 Control law design

Designing a control embedded system starts with modelling. From a model of the system which have to be controlled, control enginners define a mathematical model describing the behavior of the control system. These model equations are called *control laws.* This control law synthesis is usually based on conti-

nous time control theory. To allow their execution on computer embedded system architecures, control laws have to be expressed in the discrete time domain[1]. Inputs (sensors) and outputs (actuators) signals are supposed to be sampled synchronously at periodic time stamps. Time elapsed between inputs and outputs is not considered.

Input and output signals sampling frequencies must be chosen according to system dynamics. This choice is not unique, but too low sampling frequencies may not allow good performances while high frequencies may overload the processors. The lowest frequencies allowing system to meet requirements is the best choice. Usually, control engineers perform simulations in order to validate that the system to be controlled together with the discrete controller behave as specified. Engineers often use scientific software packages such as Matlab/Simulink or Scilab/Scicos, providing powerful computing environments suited for engineering and scientific applications design and simulation.

2.2 Control law discrete implementation

Computer engineers have to design the real time software implementing the discrete control laws provided by control enginers. This software must meet the real-time constraints, i.e. it has to guarantee that inputs and outputs are sampled periodically at chosen frequencies, and guarantee that the time elapsed between inputs and outputs is bounded.

Usually discrete control laws are implemented as *tasks*, functions with real-time execution constraints (period, deadline) and properties (execution duration, memory used,...). An execution order of a set of task must be defined according to the task's real-time constraints and properties. This execution order is called *scheduling*. Real-time computing theory provides many schedule algorithms. Execution priorities are assigned to the tasks, the highest priority task asking an execution is chosen to be executed. The scheduling may be preemptive (a task execution may be interrupted by a higher priority task) or non-preemptive. Priority may be fixed (chosen before compiling) or dynamic (computed dynamically at execution).

In many embedded systems, the schedule of the set of tasks is made at execution time by an RTOS (real-time operating system) providing a scheduler featuring preemptive scheduling.

2.3 Example : car suspension controller design

As an example of embedded system design, we focus here on a controlled suspension car system. The car body with its suspension is modelized, then a discrete control law is designed and the behaviour is simulated. Then an implementation of the control law is made.

2.3.1 The suspension control system

In this work, a seven degrees of freedom four-wheeled model (figure 1) that was adopted from [5] is considered. In this model, the sprung mass (which models the car body), is free to heave, roll and pitch. Note that roll and pitch angles are assumed to be small in order to obtain a linear model. The suspension system connects the car body to the four wheels (front-left, front-right, rear-left and rear-right unsprung masses), which are free to bounce vertically with respect to the sprung mass. A suspension element consists of a spring, a shock absorber and a hydraulic actuator at each corner. The shock absorbers are modeled as linear viscous dampers, and the tires are modeled as linear springs in parallel to linear dampers.

In order to describe this system, fifteen variables need to be considered : x_1 the heave velocity of the center of gravity of the sprung mass, x_2 the pitch angular velocity of the sprung mass, x_3 the roll angular velocity of the sprung mass, $x_{4..7}$ the deflection of each suspension, $x_{8..11}$ the velocity of each unsprung mass and $x'_{12..15}$ the deflection of each tire.

Road disturbances acting on the four wheels consist of height displacement inputs ($x_{\xi_1}, x_{\xi_2}, x_{\xi_3}, x_{\xi_4}$) and height velocity inputs ($V_{\xi_1}, V_{\xi_2}, V_{\xi_3}, V_{\xi_4}$) defined with respect to an inertial reference frame.

The suspension model has seven degrees of freedom. Consequently, only fourteen state variables are needed to describe it. The extra variable can be eliminated if the wheel deflections are expressed as a function of three state variables x_{12}, x_{13} and x_{14} and of the road disturbances x_{ξ_1}, x_{ξ_4}, x_{ξ_3}, x_{ξ_4} as illustrated in [5].

2.3.2 Active suspension control law

The control design for an active suspension in a car aims to maximize the driving comfort (as measured by sprung mass accelerations) and the safety (as measured by tire load variations) under packaging constraints (as measured by suspension deflections). In

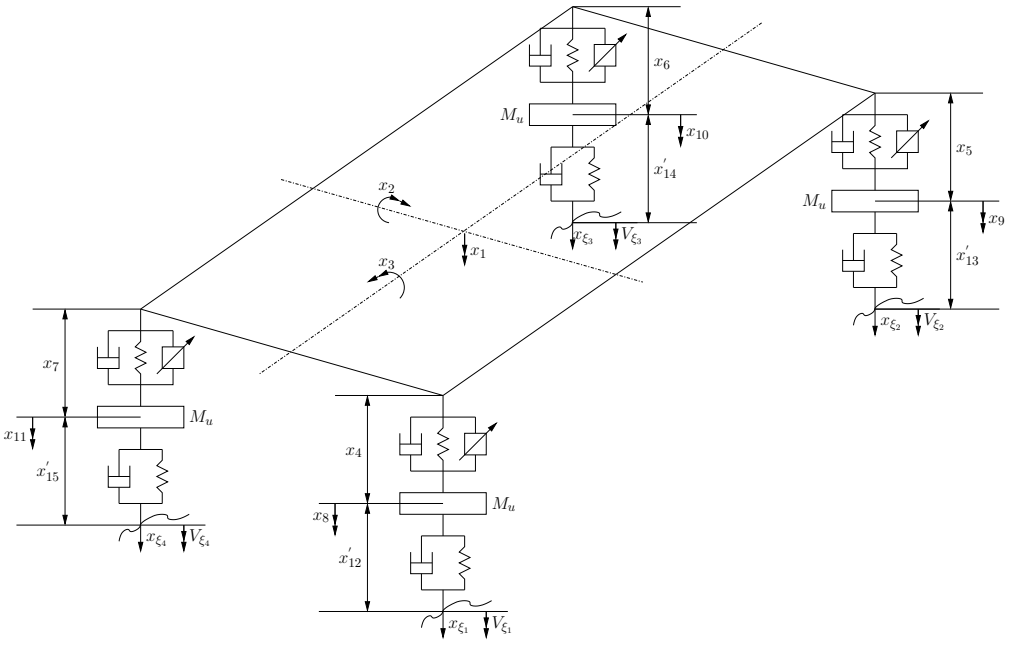

Figure 1: Full vehicule model

this paper, a suspension controller using a state feedback LQR [1] design methodology is used. The control design methodology is described in [5]. In practice, it is not possible to measure all the state variables. That's why full state observers are usually to reconstruct the state vector, from a limited number of measurements. In this paper, we assume that the suspension deflections (x_4, x_5, x_6 and x_7) as well as the unsprung mass velocities (x_8, x_9, x_{10} and x_{11}) are measured. The state vector is reconstructed using a full state observer. The controller provides the four (u_1,u_2,u_3,u_4) forces to be applied by the hydraulic actuators.

2.3.3 Simulations

In order to evaluate the performance of the designed suspension controller, the left side of the car is subjected to a "chuck hole" road disturbance [5] [2] at the speed of 40 km/h, illustrated in figure 2. Simulation results are depicted in figures 3, 4 and 5. They indicate a significant improvement in the control performance of the active suspension with respect to the passive suspension (smaller and better dumped velocities). In the following, the performance of the implementation of the suspension system will be studied. In order to evaluate it, we will only focus on heave velocity responses to the chuck hole, because the conclusions that we will get using this state variables are also valid for the other state variables.

3 How real-time implementation may impact controller performances ?

Usually, the behaviour induced by the controller real-time execution is far from the desired behaviour validated by simulation. This difference is due to the fact that simulation models usually contain simplifications of real-time implementation which doesn't match the control law design hypothesis (see §2.1): synchronous strict periodic sampling of all inputs and actuations (outputs), no delay between input sampling and actuation. This strong hypothesis lays down implicitly to design real-time implementation matching null execution duration constraints! But the discrete control law execution spends time. Thus, the real-time implementation introduce variable delays between inputs sampling and actuations and jitter on sampling time stamps. It is known that such jitter and delays impact the control laws performances [4].

3.1 Jitter

In the real-time scheduling theory, the periodic task model defines time intervals when a task should be executed: the task must start after its *activation* and must end before its *deadline*. The max time spent to execute the task in each period is the *charge* (C) of the task. The activation dates of a periodic task are equally spaced from the period of the task (T) and a deadline

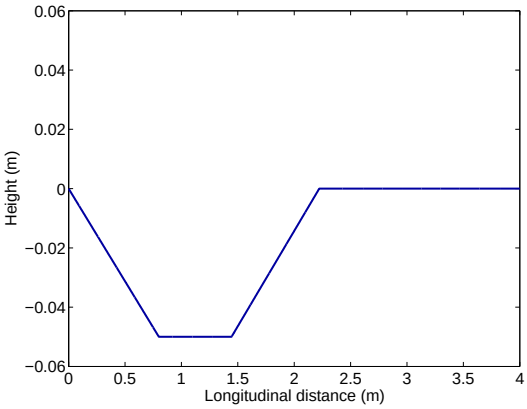

Figure 2: Chuck hole road disturbance

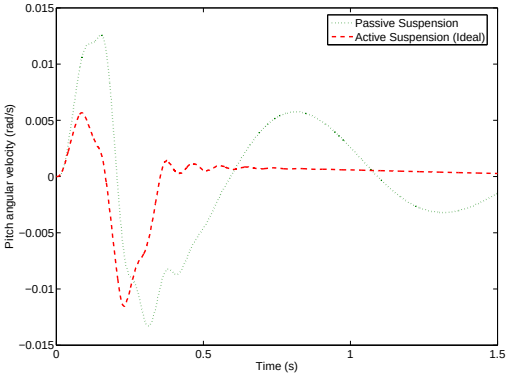

Figure 5: Pitch angular

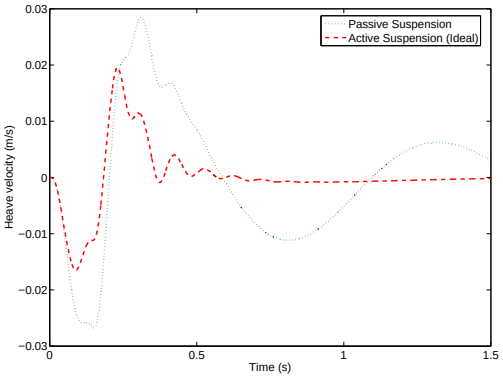

Figure 3: Heave velocity

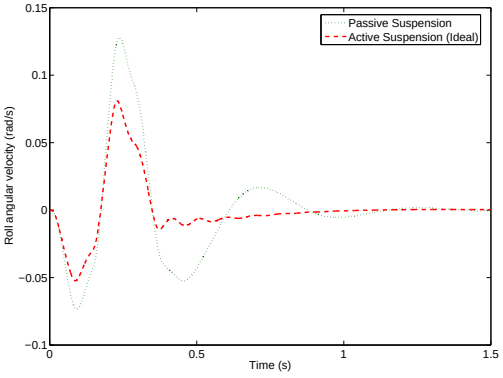

Figure 4: Roll angular

(D) is associated to each activation date. If all executions of all tasks meet these constraints, the scheduling is validated. Although the activation af a task is strictly periodic, this task model allows the task not to be executed strictly periodically because the start of the task may be delayed in order to execute a higher priority task. Figure 6 shows two different acceptable schedules of a periodic task T(C,P,D) in the usual case where D=P. We suppose that this task implements a discrete control law with an input sampling (at the begin of the task execution), some computations, and an actuation (at the end of the task execution). On the 1st execution case, input sampling and actuation are strictly periodic. On the 2nd execution case, input and output are not regular : the time elapsed between two samplings (or actuations) varies from C to 2P-C. This jitter is due to the scheduling.

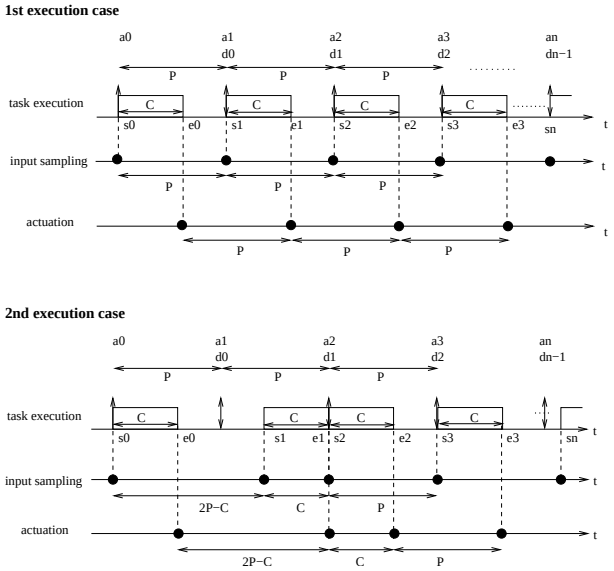

Figure 6: Jitter and delay in periodic task scheduling

Jitter may also be caused by preemption. Figure 7 depicts the same execution case as the first one of Figure 6, but here the task is preempted. The preemption delays the execution end of the task. Because the task is not preempted at each execution and because the task may not be preempted by the same higher priority task, the delay may vary. This leads to add jitter on actuation.

A scheduling is theoretically validated in the worst

execution case by taking into account the max execution duration of each task. But in the real case, a task may contain branching instructions as *if-then-else*, thus the charge of a task may not be the same at each execution. This is another cause of jitter.

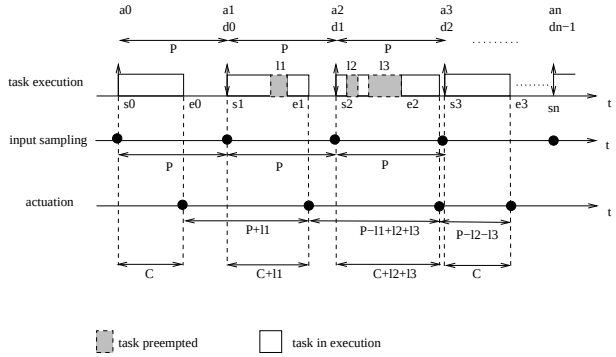

Figure 7: Jitter and delay due to preemption

Assuming that the active suspension control law is implemented in a single task, we have added a model of input sampling and actuation jitter in the simulation. The result of this simulation is shown Figure 8. Although the simulation result is closed to the ideal simulation one, it shows that the jitter inserted by the real-time execution produces a small degradation on the car suspension behaviour.

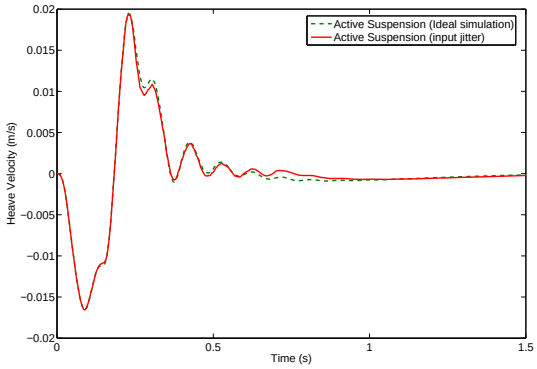

Figure 8: Suspension car behaviour simulation with jitter

3.2 Delays

In the control law synthesis step, the control engineers assume that input and actuation are produced simultaneously. This hypothesis can not be fulfilled by the real-time implementation, because a delay is inserted between the input sampling and the actuation. This delay which is not taken into account by Control Engineers may impact the control law performances.

This delay may best equal to the task execution duration (*C* on Fig.6). It is usually increased by the task preemption (Fig.7).

We have added a 7ms constant input/actuation delay in the control law of the active car suspension. The obtained simulation exhibits again a degradation on control performance.

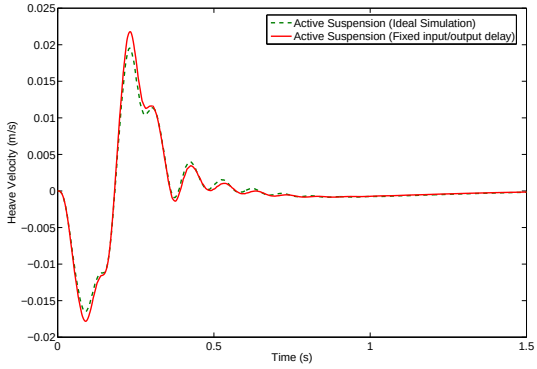

Figure 9: Suspension car behaviour simulation with constant I/O delay

3.3 Distributed implementation

In order to bring the ECU nearer the sensors and actuators and thus reduce the wiring, the control laws may be implemented on distributed architectures made of ECUs linked by communication networks. In this case, the control law may be split and the parts may be distributed on many ECUs. This distributed implementation implies network communications which may also add jitter and delays in control laws and degrade the system performances. Here, we modelize here the delays and jitters added in a control law by its distributed implementation.

3.3.1 Synchronous versus Asynchronous Communication

Data exchange between two tasks executed on two ECUs must be supplied by a communication on the network linking the ECUs. Different software mechanisms may be used to carry such a communication.

Synchronous send A task can call a communication *send* function from a communication library to generate the communication when the data is computed and ready to send. We call this mechanism *synchronous send* because it guarantees that the communication is triggered after the production of the value to send.

Asynchronous send A task can load the data to send in a buffer and ask a communication task to periodically send the buffer contains on the network. In this case, the send may be triggered by the communication task just before the refreshment of the buffer by the task which produces the data. Here, the data computation/send order is not guarantee, we call this mechanism *Asynchronous send.*

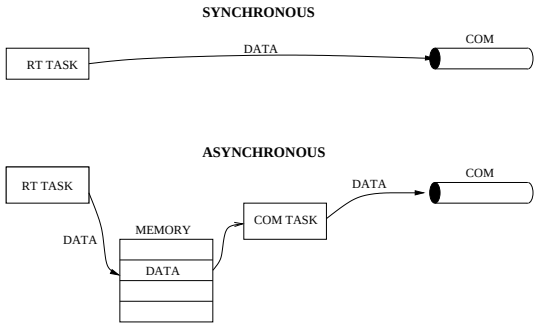

Figure 10: Send communication mechanisms

These two send mechanisms are depicted Figure 10. The receive part of a communication may also be classified as above(Fig. 11).

Synchronous receive There is a *synchronous receive* when the reception of a data allows or triggs the execution of the task which consumes the data. The data receive/consumption order is guarantee.

Asynchronous receive Here, the data receive/consumption order is not guarantee. In this case, a task may consume a data which has not been refreshed by the communication.

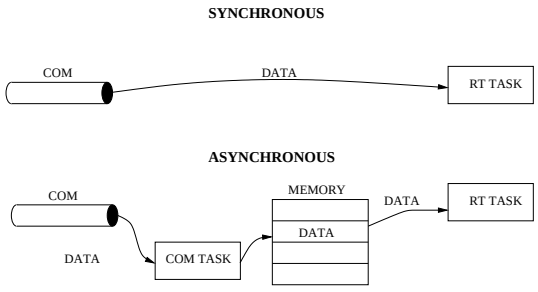

Figure 11: Receive communication mechanisms

3.3.2 Communications Models

As shown Figure 12, we can modelize 4 communications schemes combining these synchronous/asynchronous send and receive mechanisms.

As task execution, communications may insert jitter and delays between inputs and actuation. They depend on the communication scheme used. In each communication scheme, we can evaluate the time elapsed between the start of a task sending a data and the end of a task receiving and consumming this data. This delay depends on the execution duration of the two tasks, the time needed for the communication itself (communication protocol, network rate ...) and the time spent by the data in buffers (Asyn mechanisms). The tasks execution duration and the communication duration are the same whatever the chosen communication scheme. The difference is due to the time spent by the data in buffers.

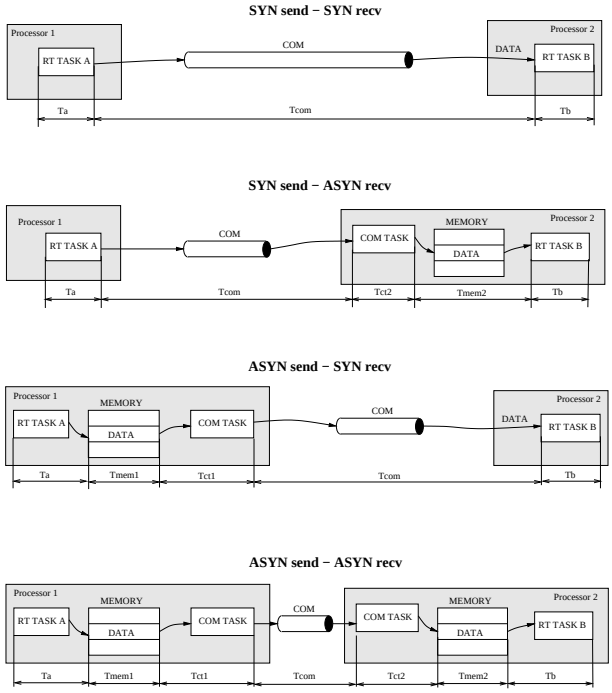

Figure 12: Communications schemes

Let's consider an example of a control law implemented as 3 tasks with the same period Te. A first task (A) implements the input sampling function. This task is executed on a first ECU. The sampled data is sent through a network (COM1) to a task executed on an another ECU. This second task (B) sends the actuation data computed (COM2) to a third task (C) executed on a third ECU. Task C implements the actuation part of the control law (Fig 13).

For each communication scheme, we will now consider the worst execution case which generates the longest delay between the input-sampling (start of A) and the actuation (end of C). To simplify this example for a didactic purpose, we assume that an ECU can not compute a task and communicate at the same time.

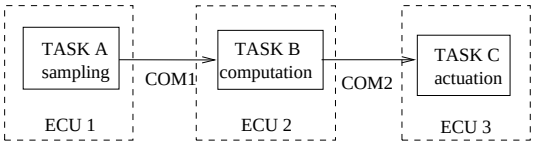

Figure 13: A control law distributed on 3 ECUs

Asyn-Asyn Communication In Asyn/Asyn communication scheme, data may be sent just before beeing refreshed and a task may read the buffer supposed to contain the data just before its reception. Executions order cannot be guarantee. Figure 14 shows the worst case schedule leading to the max input/actuation delay.

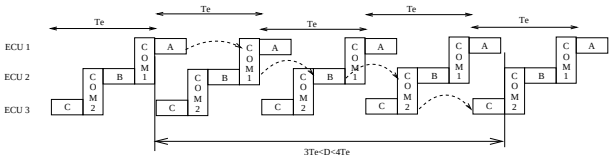

Figure 14: Worst case schedule with Asyn-Asyn communications

`C` is executed just before the communication `COM2` . Thus, the actuation is performed using the value send in the previous period. `COM2` is triggered just before `B` execution, then the data which is sent has not be refreshed yet ... In this case, the exchanged data spends nearly a time Te in each send/receive communication buffer. Consequently, the delay between the input sampling and the actuation is just over $3Te$.

Syn-Asyn Communication In this case, a data is sent just after beeing computed : the data spends nearly a time Te only in the receive communication buffer. Then, the delay is now just over $2Te$.

Asyn-Syn Communication This case is close to the previous one, but here the data spends time only in send communication buffer. Worst execution case is also just over $2Te$ (Fig. 15

Syn-Syn Communication In Syn-Syn communication scheme, no time is spent by the data in communication buffers. Thus the delay can be defined as the durations sum of each task and each communication $A + COM1 + B + COM2 + C$ (Fig.16).

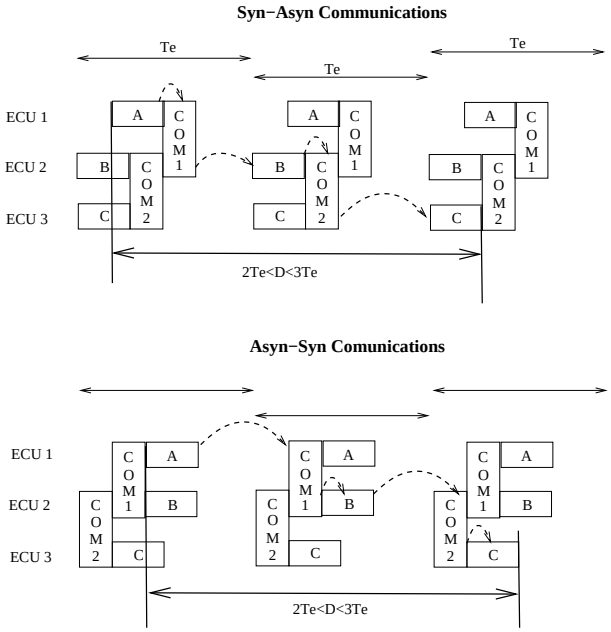

Figure 15: Worst case schedule with Syn-Asyn or Asyn-Syn communications

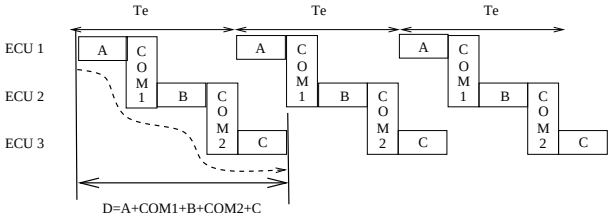

Figure 16: Schedule with Syn-Syn ommunications

4 Improving the active car suspension control law design by inserting implementation models in the simulation

We have chosen to implement the active car suspension control law on a distributed architecture composed of 5 ECUs: 4 small ECUs are based upon a small cheap microcontroller, each one is dedicated to a wheel (suspension deflection and unsprung mass velocity sampling, hydraulic actuator driving). The controller is computed on a more powerful ECU where other tasks may also be executed. Sampled data are sent to this ECU through a CAN bus (250kb/s). Sampling period Ts is 12ms. The following table sums up the execution durations of each function to be executed :

function	execution duration
Deflection sampling	0.1ms
Velocity sampling + conversion	0.2ms
Controller	2ms
Hydraulic actuator driving	1ms

4.1 Synchronous implementation

We have defined a first implementation using exclusively Syn-Syn communication. Figure 17 shows the schedule of this implementation.

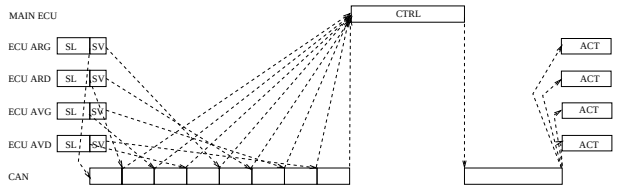

Figure 17: Implementation schedule using syn-syn communications

The results of the simulation are given by the Figure 18. The behaviour is less good than the one predicted by the "ideal simulation" but still acceptable.

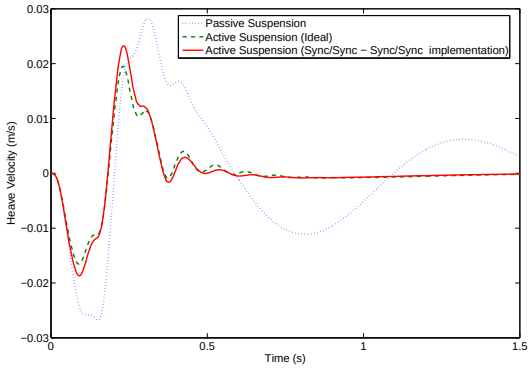

Figure 18: Synchronous implementation simulation result

4.2 Asynchronous implementation

The 4 small ECUs are realized by a supplier. The controller is implemented by the car manufacturer on a more powerful ECU already used for other functionnalities. The software executed on the small ECUs is written by the supplier following the car manufacturer specifications. In this case, it may be difficult to synchronize computation tasks and communications in order to guarantee Syn-Syn communications. This leads to choose a non-synchronous implementation easier to set.

On the 4 small ECUs, strong cost constraints enforces not to use RTOS services. Thus, the implementation is based upon the periodic execution of the following sequence: *deflection sampling*, *velocity sampling*, *sending the deflection and velocitie values* on the CAN bus, *Hydraulic actuator driving* (Synchronous send). Each 12ms a timer triggers an interruption which start the execution of this sequence. The reception of a CAN message containing the *actuation value* triggers an interruption which stores the received value in a buffer (Asynchronous receive). On the fifth ECU, the controller is implemented as a task handled by an RTOS. CAN communications are handled by a specific task (Asynchronous send and receive). This second implementation has been modelized using Syn-Asyn and Asyn-Asyn communication mechanism models.

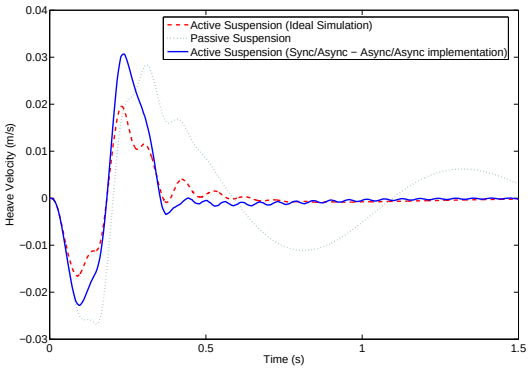

Figure 19: Asynchronous implementation simulation result

The simulation (Fig.19) shows that the impact of this new implementaton on the control behaviour is significant and it may not be acceptable. To enhance the performances of this second implementation, Control Engineers may choose to reduce the sampling and actuation period. The simulation shows that a control law discretized with a 3ms period allows to obtain performances close to the synchronous implementation ones (Fig.20). But, the real-time execution of this new control law requires 4 times more processor and network ressources.

5 Conclusion and work in progress

In that paper, we have described how real-time implementation mechanisms (computation durations, scheduling, communications,...) could generate delays and jitters which degrade the control law performances. Adding implementation models in simu-

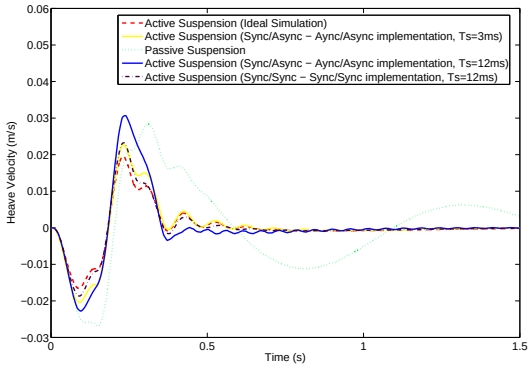

Figure 20: new control law simulation result

lations allows control engineers to take into account this degradation and helps them to better tune the control law parameters. We have illustrated this approach by an active car suspension controller design example. On the example, this enhanced simulation has shown that a synchronous distributed implementation required less resources at execution than an asynchronous one. Here, implementation models must be described and added to the control law manually but we are currently working on a tool that can do it automatically. We hope, by this way, to reduce significantly the design lifecycle of control embedded systems.

References

[1] ASTROM, K. J., AND WITTENMARK, B. *Computer Controlled Systems: Theory and Design*. Prentice-Hall International, 1984.

[2] BEN GAID, M., CELA, A., AND KOCIK, R. Distributed control of a car suspension system. In *European Simulation Congress on Modelling and Simulation (EUROSIM 2004)* (ESIEE Paris, 2004).

[3] CERVIN, A. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, Apr. 2003.

[4] CERVIN, A., HENRIKSSON, D., LINCOLN, B., EKER, J., AND ARZEN, K.-E. How does control timing affect performance ? *IEEE Control System Magazine 23*, 3 (2003).

[5] CHALASANI, R. M. Ride performance potential of active suspension systems Part II: Comprehensive analysis based on a full-car model. In *Proceedings of the Symposium on Simulation and Control of Ground Vehicles and Transportation Systems, ASME AMD* (Anaheim, CA, Dec. 1986).

[6] KOCIK, R., AND SOREL, Y. A methodology to reduce the design lifecycle of real-time embedded control systems. In *ESM'2004* (Paris, October 2004).

[7] MARTÍ, P. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. PhD thesis, Automatic Control Department, Technical University of Catalonia, Spain, 2002.

[8] PALOPOLI, L. *Design of Embedded Control Systems under real-time scheduling constraints*. PhD thesis, ReTiS Lab - Scuola Supoeriore S. Anna, Pizza Martiri della Libert´a, 33, Pisa Italy, July 2002.