Signal Processing in Resource Insufficient Environment

Sebestyén Bartha, Zoltán Gábriel, Lajos Mezőfi, Gábor Péceli
Department of Measurement and Information Systems, Budapest University of Technology and Economics, and
Embedded Information Technology Research Group, Hungarian Academy of Sciences, Budapest, Hungary
Phone: +36 1 463-2057, fax: +36 1 463-4112, e-mail: peceli@mit.bme.hu

Abstract ***– Most embedded signal processing applications are developed in at least two separate stages: signal-processing design followed by its digital implementation. With such an approach computational tasks that implement the signal processing algorithms are usually scheduled by treating their execution times and periods as unchangeable parameters. Task schedulability therefore is independent of the actual state of the physical environment; it depends only on the amount of computing resources available. In embedded systems, typically due to power and energy constraints, the available computing resources are definitely limited. A better overall performance might be achieved if signal-processing design and task scheduling are linked, and an integrated approach is applied. In this paper an attempt is made to handle temporary resource insufficiency by introducing Quality-of-Service (QoS) adaptation into signal processing. The approach applied can be considered as a "never-give-up" strategy, where the signal processing is performed in any case at the price of lower quality. In the proposed solution different algorithms are available at task execution level, having different execution times and quality. The version to be executed is selected by the ongoing scheduling mechanism. In our experimental setup the Earliest Deadline First (EDF) algorithm is applied for this purpose, and different-order median-filters are utilized to illustrate the concept of QoS adaptation in signal processing.***

Keywords ***– QoS adaptation, flexible scheduling, mode adaptation, adaptive signal processing and control***

I. INTRODUCTION

This paper reports a recently started research and development project aiming at flexible real-time execution schemes and environment for embedded signal processing and control applications. The expected output of this project is on one hand a series of new services, which enhance the capabilities of the latest embedded operating systems, while on the other a new development environment, which offers efficient tools and methods to prototype distributed real-time embedded systems.

The addressed schemes can simultaneously handle hard and soft real-time task executions, and tasks with non-critical resource utilization. Depending on the nature of the embedding environment the tasks can have several modes of operation, which are expressed by different sets of periods, execution times and deadlines. The scheduler is dynamically informed which set is valid for the next run. The tasks may dynamically indicate which task should be co-executed as a direct consequence of the actual results of the control law or signal evaluation.

For the resource insufficient situations a different flexibility is provided. It is based on the fact that signal processing and control law evaluation can be performed at different QoS levels. Again the tasks have different modes of operation, but the version to be used at a given point of operation is selected by the scheduler based on its actual knowledge of the available resources. Obviously the QoS level of the overall system cannot be lowered for longer time. It is the responsibility of the task, or of a higher lever supervisory mechanism to counteract if a permanent QoS degradation is observed. If the system resources are still available, the task can limit the application of the lower QoS levels, and implicitly force some other tasks to temporarily reduce QoS level, if necessary. All the above concepts are planned to be extended toward sporadic tasks, as well.

QoS adaptivity is a well-known concept in communication engineering [1], but it is relatively new in the field of embedded information systems. This is due to the natural conservatism of control engineers, especially in the case of safety critical applications [2], where the unavoidable dependability requirements are met more easily by design-time decisions, and typically at the price of minimum flexibility in execution time. Quite recent research papers, however, are arguing for the run-time QoS adaptivity of

Supported in part by the DARPA's Software-Enabled Control Program (AFRL contract F33615-99-C-3611), and by the Hungarian Ministry of Education (OM-FKFP 0654/2000).

0-7803-7864-4/03/$17.00 ©2003 IEEE.

different real-time services in embedded systems ([3]-[6]). Such flexible embedded solutions become more and more realistic mainly due to recent advances in microprocessor and micro-controller technology.

The purpose of our efforts is to establish a new generation of flexible and reliable real-time scheduling mechanisms mainly for signal processing and control applications. These mechanisms serve as parts of a general background infrastructure for implementing higher-level adaptivity services like fault adaptation, reconfigurations and mode changes, transient management, etc. [7].

To enable experiments with an operating system having QoS adaptive dynamic scheduler, the modification of a simple, but efficient embedded operating system (called µC/OS [8]) was decided. This modification is introduced in Section II. Section III is devoted to summarize the key components of the necessary development tools, while Section IV reports the testing of the adaptive scheduler. Finally Section V presents some considerations and an illustrative example concerning resource-adaptive signal processing.

II. ADAPTIVE DYNAMIC SCHEDULING

This section describes the parameters, the scheduling algorithms, and the most important new functions of the modified version of µC/OS. This software component serves as the core part of our experimental setup.

A. Input Parameters of the Scheduler

The operating system uses the following task parameters:
Priority: This value sets the initial priority of tasks. Its role is only administrative, because while running the operating system dynamically swaps the priorities of tasks. In the operating system each task *must* have a valid priority, which must be between 0 and 62 (zero being the highest).
Period: The operating system can handle periodical and non-periodical tasks. In the case of periodical tasks, this value sets the time between two periods. It is given in operating system time ticks (ticks). When declaring non-periodical tasks, the period value must be zero.
Deadline: This value represents the time until which the task must finish its actual job. It is also given in ticks and is relative to the starting time of the task. The deadline must not be longer than the period, but it is preferred to be shorter if possible.
Computation time: The computation time is the time required by the task to finish its current job. It is given in ticks and is relative to the starting time of the task. The computation time must not be longer than the deadline, and similarly to the previous value it is preferred to be shorter if possible. This also means that the computation time must be shorter than the period. The modified operating system can handle up to three different computation times for a task. These computation times represent different qualities of processing, and the operating system chooses the appropriate one according to the current CPU load. Less than three computation times can be set by giving zero value to the ones not needed.
Offset: The tasks do not necessarily start at the same time. With the offset value the starting time of a task can be set. This value is given in ticks. The offset can also have a zero value, which means that the task starts right when the operating system starts.
These parameters should be given as constants in the source code of the corresponding tasks. The operating system will get these values via the OSTaskCreate() call, and stores them in Task Control Blocks (TCB). As it is presented later, the values can be set via a graphical user interface, which saves them in the format required by the operating system.

B. Algorithms used in Scheduling

The most important algorithm used is the Earliest Deadline First (EDF) algorithm. This algorithm - in contrast with the original scheduler of the operating system - does not use priorities but runs the tasks based on their deadlines. It is always the task with the earliest deadline that is run by the scheduler.

To be able to do so, the scheduler must sort the tasks based on their deadlines, and choose the one with the earliest deadline. The original version had a simple priority based scheduler, which was not able to deal with the above requirements. With a little modification however it is possible to transform it into a dynamic scheduler. The scheduler must sort the tasks based on their deadlines before scheduling. The chosen algorithm for that was the bubble sort algorithm. In the next step the algorithm compares the second ready to run task with the rest, and so on. In the end the task with the earliest deadline will also have highest priority. This means that from here the original scheduler can be used without modification.

The order of the tasks changes only when a new task becomes ready to run or when the current task finishes its job. In these cases a rescheduling (which also means a new sorting) is needed. In this new concept the running of a task can be interrupted if a new task becomes ready to run with an earlier deadline. It can be seen that the initial priorities do not play part in the scheduling, because before the scheduler would use them, the sorting instantly changes them. And although the tasks have priorities, they are scheduled based on their deadlines.

If a task has more than one computation time, the operating system must decide which one to use. And even if there were only one computation time it would be good to know if there is enough CPU time available before its deadline. The run-time algorithm that is used to check this is called immediately after the occurrence of a new request, and it decides in which mode this new task should run. As a first attempt a very simple heuristic algorithm was implemented, which checks only a very simple necessary condition of scheduling. As it is highlighted in Section IV, based on test

results, further refinements are needed to improve the "success rate" of the proposed adaptive dynamic scheduling algorithm.

As a first step this algorithm collects all information about the tasks that should start running between the current time and the deadline of the actual task (the one whose computation time we want to decide). It sums the computation times of these tasks. Then it selects the latest deadline among these tasks. If the remaining time until that deadline is not less than the sum of the computation times of the tasks, then the new task can run with the selected computation time. If the sum of the computation times is larger, then the same algorithm is replayed assuming an execution having shorter computation time (if available). The process goes until the task can fit into the time available. (Using this simple algorithm the mode of the other tasks will not be changed.) If none of the computation times can fulfill that, then the task has to be dropped.

C. The Structure of the Modified μC/OS

This sub-section describes the structure of the new scheduler (see Fig. 1), and the most important new functions of the modified operating system.

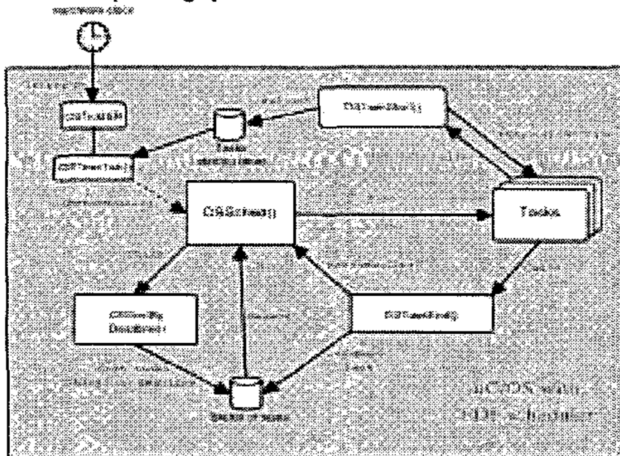

Fig. 1. The structure of the modified μC/OS

OSTaskStart(): Each task has to call this function at the beginning of each of their runs. The first thing the function does is to calculate the time when the task starts to work next. This is only done for periodical tasks and it is the sum of the current time and the task's period. Next it selects the appropriate computation time with which the task should run. If there is no computation time that would fit in the remaining time then the task is dropped. The algorithm used here is described in the previous section. The function returns an integer value with the number of the computation time that should be used (a special value, OS_TASK_DROPPED is returned if the task is dropped).

OSTickISR(): This is the interrupt handler routine that is called whenever the computer's hardware clock sends an interrupt. It's only task is to call the OSTimeTick() function. This function makes it possible that the scheduling can be done in real time. This function was not modified; it is in its original state.

OSTimeTick(): This function is called after the tick of the hardware clock. Its original function was to check whether a task's delay is finished or not and to increase the inner clock of the operating system. Besides these, the function now compares the current time with the time of the tasks' next run. If there is a task that should start then the function clears the task's suspended state and sets it as ready to run. After this it calculates the task's absolute deadline, that is the sum of the current time and the task's deadline. It is used in the sorting of tasks based on their deadlines.

OSSched(): This is the actual scheduler. It selects the task with the highest priority and runs it. The only modification that was done is that at every run it calls the function which sorts the tasks (OSSortByDeadline()).

OSSortByDeadline(): This function sorts the tasks based on their deadlines. The algorithm used here is described in the previous section.

OSTaskEnd(): This function must be called at the end of each task's run. It clears the absolute deadline of the task and suspends it. The next time the task is run will be at the time that was set by the OSTaskStart() function.

III. DEVELOPMENT TOOLS

This section describes the functionality, abilities and usage of the Graphical User Interface (GUI), which supports the parameterization of the tasks, the automatic code (task-skeleton) generation and the monitoring.

A. Description, and Usage of the GUI

The graphical planning tool is meant to aid the development process by giving a more comfortable interface for task parameterization, code generation and monitoring. A screenshot of the tool can be seen on Fig. 2.

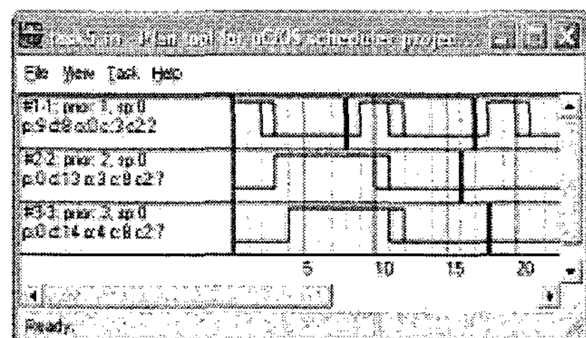

Fig. 2. The planning tool

Menus and dialogs are used to control the tool. The menu items' descriptions, thus the available functions of the tool and their usage can be read below.

The actual task list can be saved or cleared, and previously saved task lists can be loaded with the appropriate *File* menu items.

View menu:

- *Display settings* – the displayed width of the time ticks, the maximal displayed tick's number, and the list of the displayed tasks (all tasks are shown by default) can be set.
- *Zoom in/out* – multiply/divide the width of ticks by two.
- *View schedule results* – the log file of the current task list's run (if exists) can be loaded.

Task menu:

- *Add task* – add a new task and set its parameters: a short description, periodicity, number of computation time alternatives, computation times (in decreasing order), deadline, offset and initial priority.
- *Modify task/Remove task* – modify or remove tasks
- *Generate tasks' skeletons* – generate source from the actual parameters. This option's further description can be found in the automatic code generation section.

The current task list is displayed on the main canvas in a table. Each task occupies a row. The first column contains the parameters of the tasks in a text form – each parameter is indicated with its first letter. The second column contains the diagram of the ideal timelines of the tasks. The line's high state represents running state, while the low state represents the not running state. All three computation-time versions are displayed on the same diagram. The thick lines represent the deadlines for the previous occurrence.

If the log file has been loaded, the first two rows of the table show the results. The first row contains the system's timeline and the second shows the dropped tasks. The system's timeline is similar to the tasks' timelines, except that all tasks are represented in a single diagram. To be able to know which task is currently running, the ID of the task, the chosen computation version's number and the number of the current run is written on the diagram, above or under the low-high state transitions (syntax of the displayed text: <task ID>.<comp. version>.(<number of run>)).

B. Automatic Code Generation

The tool can generate a C source from the actual parameters (deadline, offset, computation times, etc.) of the planned tasks. The source takes place in a single C file, which contains the skeletons of the tasks and the configuration of the operating system.

Each task is represented with a function, which contains an infinite loop and a four-way branch. Within this loop, one case stands for each computation algorithm version, and one for the case the task cannot be scheduled, thus has to be dropped. Being a skeleton, this only contains the format required by the operating system; the user must add the contents of each case. The functions always contain three cases for the different computation methods. If not all three is needed, the computation time of the obsolete cases must be set to zero.

The code generator writes a function for each task in the source file, and their parameters are written in *#define* lines. It also writes a main function which contains the redirection of the interrupt vector to the μC/OS, the initialization of the operating system, the registration of the tasks' functions (the *OSTaskCreate* function with the corresponding parameters must be called for each task), and the calling of the *OSStart* function which starts the operating system and never returns.

There is an option in the generation to choose between test mode and normal source. When choosing test mode, the source is extended, so that each task writes to a log file the following: which computation algorithm was chosen, when it started computing, and when it finished. The results taken from the log file can be loaded and graphically represented on the GUI.

C. Monitoring

The monitoring/logging functionality is vital to be able to test the correctness of the system. Thus, the plan tool designed for aiding the development process also supports this functionality. Collecting data from the running system can be done in two ways: collecting data to a log file (and analyze after the system stopped) or monitoring from a separate computer via a monitoring task. Of course each method affects the system performance, but there are no other efficient ways for monitoring. Currently only the first method is supported.

The second method offers real-time monitoring, but the collected data is the same with both methods. This includes: the selected task's ID, this task's selected algorithm's ID, the dropped tasks' ID and system time. The data is drawn on a diagram on the GUI. This diagram shows a timeline of the system's activity, where the line's high state represents activity, and the low state represents inactivity. In addition, statistical information is calculated, such as processor usage, ratio of dropped tasks, tasks that never get processor time, etc. The statistics can be used to calculate optimization for the tasks' parameters if possible – for example giving different offset to the tasks, so that processor usage, and dropped tasks ratio improves.

IV. TESTING OF THE SCHEDULER

As it was mentioned in Section II, to demonstrate the concept, first a very simple scheduling algorithm was implemented. The refinement is planned to be based on the results of intensive testing.

While we are testing, we have to verify, among others, the following properties: the scheduler lets execute as many tasks as possible; if a task could run in high-quality execution mode, then it should be avoided, that the scheduler forces this

task to reduce execution time; let no task run always at the lowest level execution mode, and so on. So we have lots of considerations, which all need to be kept, if we want to have a correct and near optimal scheduler for real time problems; for this reason test vectors were constructed, which "study", how the scheduler behave in certain situations. Knowing what are the optimal solutions in all situations (the test vectors), we can compare the results of the program to our own theoretical scheduling. If there is a difference between the two solutions in a testing situation, and the programs result seems to be worse, then we have a situation, where the scheduler doesn't function properly. Collecting these cases, we can locate the problems in the source code or in the used algorithms and after that, we have to correct them.

After creating about 40 test vectors and verifying the program, we have found some unacceptable failures, which are not corrected yet, because they need further re-thinking of some aspects of the scheduling. These are detailed in the following paragraphs, and are named as: (1) local scheduling problem, (2) „first tasks fall out" problem, (3) periodic task problem, (4) real time scheduling problem.

Local scheduling problem means that the scheduler examines schedulability only for a single time interval, which is determined by the latest deadline of one of the tasks until all tasks ready to run must be executed (so called outer deadline). The scheduler adds together the execution times for every task, and if this sum is below the outer deadline, then the scheduling can be done with these execution times, otherwise it must reduce the execution times or drop one ore more tasks, until the sum is smaller, than the outer deadline. The problem with this method is, that it can happen, that in a smaller time interval the tasks cannot be scheduled; there are too many of them, and all should be executed until a special deadline, even before the outer deadline. So some of the tasks cannot be scheduled, but the scheduler "is convinced" they can. A possible solution for this can be, when we re-schedule the tasks at each incoming task, and we always see the newest tasks deadline as the outer deadline (it is true, for the solution we have to pay with time).

To illustrate the local scheduling problem lets investigate a system consisting of 9 tasks, having execution times: 2,8,2,2,2,3,3,3,8 and different offsets (starting phases), which are not detailed here. Figure 3 stands here to illustrate the scheduling situation: task2, task8 and task9 (not indicated in the figure) have deadlines at 22, 18, and 22 ticks, respectively. The sum of the execution times is 33 ticks, but the "last" task should be executed at time t=32. The implemented algorithm decides to shorten one of the tasks with 1 tick. However, this is not a correct solution, because there are other critical deadlines at 18 and 22 ticks.

Another remaining problem is, that the „first tasks fall out". That is because only one task can be modified at a time, so only one task's execution time can be reduced in a step. The program always starts with the first incoming tasks: these will be changed to lower quality mode or even dropped, to take the sum of all execution times below the outer deadline.

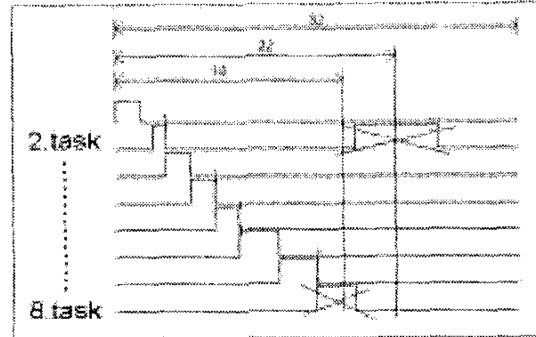

Figure 3. Illustration of scheduling problems

The scheduler tries to reach a schedulable state now at the first task, and it doesn't realize, that a scheduling can also be made, when we don't harm the first task, and instead try to change the later tasks. It is obviously bad for us, but we will able to replace this method with another one, which possibly will need more processing time.

The handling of periodic tasks is also not built in yet, but it is only a matter of time. The current version only counts with the first ready-to-run part of each periodic task. Here the critical point is to find an outer deadline, even critical if we have more periodic tasks with different periods. But this is a problem can be handled if we can compute a „long period", which is the time, when we reach the starting state again with the periodic tasks.

A fourth problem is that the scheduler currently operates only in a simulation environment, and therefore real-time measurements are not available yet. Having the complete description of tasks, the current version of the scheduler finds the latest deadline of the tasks, and makes the scheduling to this date. A real time version could make the scheduling in any time to the latest deadline of the tasks, which have already appeared until the passing time. What causes further complications is, that in real time environment the scheduler doesn't know anything about the tasks having not showed up yet. We need some a priori information, or the scheduler don't know, that at an incoming task whether it can fill in the available time or should wait for an other task, which hasn't yet appeared.

V. ILLUSTRATIVE EXAMPLE

Imagine a system with different operational modes. Figures 4-6 show signals in three operational regions, where different processing algorithms, having different execution times, might be required. The simple problem is to reduce the effect of impulse noise by median filtering. The available computational power in the second region is supposed to be limited; therefore here a reduced quality median filtering is applied (7-points median filter in the first and the third regions, while 3-points median filter in the second one). Investigating Fig. 6 it is obvious, that in the second region the

noise reduction is almost minimum, while in the other two quite acceptable.

The proposed QoS adaptation can be utilized with every recursive signal-processing algorithm, which follows the so-called prediction-correction scheme, because temporarily the evaluation of the correction term can of lower quality without affecting the overall performance seriously.

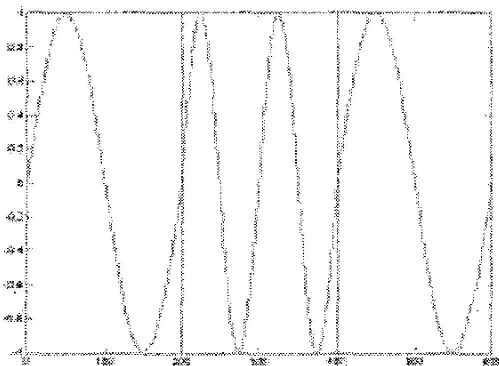

Figure 4. Sinusoidal waves in the three different operational regions

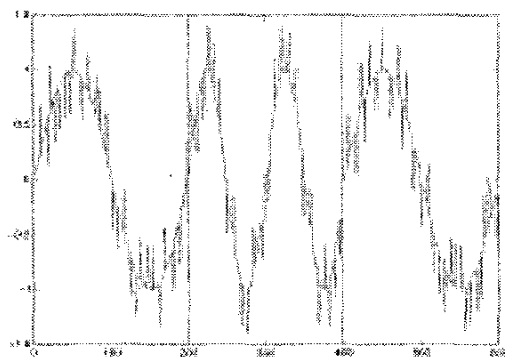

Figure 5. Sinusoidal waves + impulsive noise

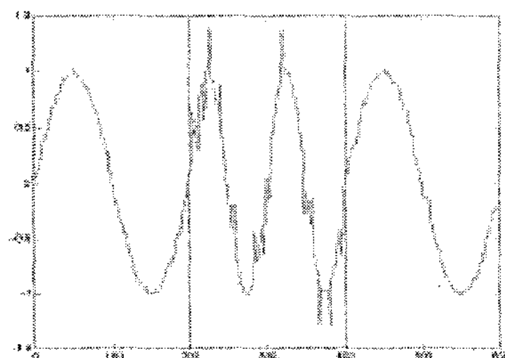

Figure 6. Median-filtered time-sequences in the three different operational regions

VI. CONCLUSIONS

The introduction of QoS adaptivity into the implementation of signal processing and control can considerably improve resource utilization at the prize of occasional quality degradations. The power and energy constraints in embedded systems ask for serious design considerations in several respects. The proposed approach suggests a cooperative design procedure, where signal processing and control designs are interacting with that of resource utilization and scheduling in hard real-time systems. An other important aspect of the proposed contribution is, that it opens or widens research areas in modern systems engineering, where mode adaptation and mode control seems to be a rather promising compromise to handle complexity and assure robustness.

REFERENCES

[1] M. Metso, A. Koivisto, J. Sauvola, *"Multimedia adaptation for dynamic environments"* in *Proc. IEEE Signal Processing Society Workshop on Multimedia Signal Processing,* Vol. 02, pp. 203 - 208, December 1998.

[2] H. Kopetz, *Real-Time Systems; Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[3] C. Lu, J.A. Stankovic, S.H. Son, G. Tao, "Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms," *Real-Time Systems*, Vol. 23, No. 1-2, pp. 85-126, July-September, 2002.

[4] D. Seto, J.P. Lehoczky, L. Sha, K.G. Shin, "Trade-Off Analysis of Real-Time Control Performance and Schedulability," *Real-Time Systems*, Vol. 21, No. 3, pp. 199-217, November, 2001.

[5] R. Chandra, X. Liu, L. Sha, "On the Scheduling of Flexible and Reliable Real-Time Control Systems," *Real-Time Systems*, Vol. 24, No. 2, pp. 153-169, March, 2003.

[6] D.C. Schmidt, "Adaptive and Reflective Middleware for Distributed Real-Time and Embedded Systems," in *Proceedings of the 2nd International Conference on Embedded Software, EMSOFT'2002,* (eds. A. Sangiovanni-Vincentelli, J. Sifakis) Grenoble, France, October 7-9, 2002, Springer Verlag, pp. 282-293.

[7] G. Simon, T. Kovácsházy, G. Péceli, "Transient Management in Reconfigurable Systems," in P. Robertson, H. Shrobe, R. Laddaga (Eds.), *Self-Adaptive Software,* Lecture Notes in Computer Science, Vol. 1936, Springer, 2000.

[8] Jean Labrosse, MicroC/OS-II, The Real-Time Kernel, 2nd Edition, CMP Books, 2002. URL: http://www.ucos-ii.com