Applying Fixed Priority Scheduling in Practice

Raimondas Lencevicius, Alexander Ran
Nokia Research Center
5 Wayside Road, Burlington, MA 01803, USA
Raimondas.Lencevicius@nokia.com Alexander.Ran@nokia.com

Categories and Subject Descriptors
D.4.1 [**Process Management**]: Scheduling.D.4.8 [**Performance**].

General Terms
Measurement, Performance.

Keywords
Software architecture, scheduling

1. INTRODUCTION
Last year our research group was requested to study the runtime architecture of mobile phone software in order to understand whether some performance aspects of the phone could be improved. We expected to improve the architecture by systematically deriving task priorities for more effective scheduling and improving partition into tasks.

A major part of the runtime architecture improvement work was concerned with the scheduling of the tasks in the mobile device. We were aware of the large body of research work in fixed-priority scheduling area (we reference a small subset [2]-[5],[7]). However, when we tried to apply this research in our real-world situation, we discovered a number of problems described below.

In this paper we are primarily concerned with embedded real-time systems such as mobile phones or personal communication devices. Today personal communication devices are more than voice call terminals. Mobile phones serve as platforms for a variety of mobile applications including text and picture messaging as well as personal information management, including data synchronization with remote servers and desktop computers. Mobile phones host a range of communication-centered applications most of which have real-time constraints. During a voice call speech data must be processed in a timely fashion to avoid jitter. During a GPRS session lower layer packets arrive every 10 ms. These are just few examples of system requirements that lead to tight software performance constraints.

To improve the performance of the mobile phone software, we concentrated on the runtime software architecture—a partition of all software functions into concurrent units and a scheduling policy that deliver the best possible service to the user with available resources. The units of concurrency in most products are operating system tasks. Thus partition of software into tasks and allocation of functionality to tasks in the form of objects or functions are the most important decisions in the design of the runtime architecture [6]. In this paper, we focus on the scheduling policy and its parameters.

2. FEATURESETS
Many of the applications on a mobile phone may be executed concurrently but not all. In fact, it is impossible to execute all the applications concurrently due to conflicts and contention over the use of specific hardware resources on one hand and timeliness and other quality constraints of the applications on the other. It is then essential to identify the sets of applications that are concurrently useful and investigate whether it is possible to execute these sets concurrently on given hardware. Thus an essential concept in the mobile device runtime architecture is a *featureset*. A featureset is a set of concurrently available features. Specification of useful featuresets for a given system is an architecturally significant requirement. All featuresets have to be schedulable. A featureset is defined by a collection of use cases that can overlap in time. There are scenarios that correspond to these use cases. Objects participating in these scenarios can be concurrently active. A featureset determines which objects may need to be executed concurrently. All objects are allocated to some task. The tasks that contain objects from the same featureset have to be collectively schedulable (Figure 1).

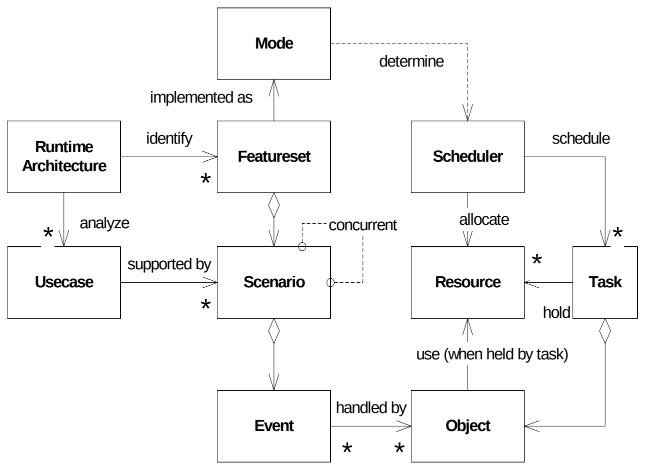

Figure 1. Runtime Architecture Meta Model

Tasks that do not belong to the same featureset do not have to be collectively schedulable. If tasks that belong to the same featureset are not schedulable, some architectural decisions regarding allocation of objects to tasks or resource scheduling policies must be revised.

Featuresets are a system design pattern. System designers identify featuresets by first composing a collection of all important use cases. Then designers identify subsets of this collection containing use cases that may overlap in time. Such subsets contain concurrently available features—featuresets. Now a mapping between featuresets and tasks is needed. Each use case contains one or more execution scenarios. Each scenario is defined by one or more sequences of events that occur in the scenario. One or more objects handle each event. Each object is allocated to one of the concurrent tasks. Therefore, each featureset has a mapping to a set of tasks (Figure 1).

Since each featureset maps to a set of tasks and no other tasks can run in this featureset, it seems natural to analyze each featureset separately when assigning task priorities. In this case, different schedules are used for different featuresets. Such priority assignment is called in fixed priority scheduling a *mode* [5][7]. Although featuresets can be implemented as modes and mapped one-to-one to their modes, featuresets differ from modes in several aspects. Featuresets are a practical way based on system runtime architecture of finding a minimal set of events or tasks that need to be concurrently schedulable. This allows to minimize the number of tasks and therefore the number of real-time constraints in a mode. Such mode design becomes more stable—additional tasks and constraints can be added to the mode in the future without breaking the schedulability. If modes were designed as sets of concurrently schedulable tasks or events without the consideration of the featuresets, addition of new events in the course of system evolution would often require mode redesign. The need for mode minimization and our approach to minimization are not discussed in the scheduling literature.

3. TASK MODEL FOR SCHEDULING

Whether we are scheduling tasks in a featureset or a whole system, we need to know certain properties of tasks and resources to create a schedule. Although there are a number of interesting and effective approaches for dynamic scheduling of tasks [5], most industrial real-time systems today still rely on static fixed priority scheduling. What information is needed to assign task priorities in fixed priority scheduling? We need to obtain task periods, execution times, and deadlines or constraints on response times. In addition, currently assigned task priorities need to be known too. This is the minimal information needed for making effective scheduling decisions. In the case of our system, we had to produce a list of concurrent tasks and for each task to determine its priority, period, execution time, and deadline or constraint on response time.

We had full access to the design documentation, the source code, and extensive execution traces produced from multiple use cases. Unfortunately, the design documentation often does not contain sufficient information regarding the task structure. This is mainly because the initial task structure is often changed at later stages of product integration and fine-tuning when the design documentation is not actively maintained anymore. Although the source code does contain task creation instructions and task priorities, it is not easy to determine from the source code which objects and functions are allocated to which task and how the objects in a given task interact with objects in other tasks. Therefore we mainly focused our efforts on the analysis of execution traces, which proved to be the easiest way to obtain the needed information.

Mobile software is instrumented to produce a variety of traces that can be enabled or disabled at runtime. In our study we relied on the traces produced by the operating system scheduler and inter-task messaging. A trace record is produced every time a task is scheduled to run.

The raw data we started with consisted of multiple files of traces collected for different use cases. A typical use case is a file download over a local connectivity interface, e.g., USB, during a voice call. Another example is using a mobile phone as a cellular modem for downloading a file from a laptop over Bluetooth and simultaneous upload over GPRS to a remote server.

Task activity attributes like period or execution time should not be calculated over the entire duration of the use case. This gives incorrect information because most tasks are not active during the entire duration of the test case. The trace file usually includes traces from different stages of the process like connection setup and teardown, buffering of the data downloaded over Bluetooth before starting the GPRS upload, and so on. Thus we had to determine the relevant segment for trace analysis. Although we considered a variety of automatic techniques, the most efficient way was to visually examine the utilization map and to identify the segment of the use case that has all the relevant tasks active.

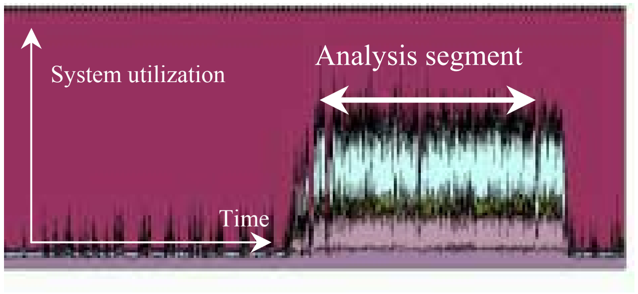

Figure 2. Finding the analysis segment from utilization map

A utilization map is a graph we produce from an execution trace file. It shows the processor utilization by tasks as a function of time. Each task's utilization is color-coded and the total utilization is represented as a histogram over the time interval (Figure 2). It is quite easy to identify the segment of the use case where all tasks that have to be scheduled are active. We use this information to discard the traces contained in a trace file that fall outside of the selected segment.

Task period is defined as the interarrival interval of events causing task's execution. Since such event traces were not available in our case, we approximate the task period by the time interval between task's successive invocations. If a task is scheduled later than the time of its event arrival, the previous task period will be overestimated and the next one underestimated. The execution time of a task can be calculated as the time between an invocation of a task and the invocation of the next task. This approach is correct only if tasks run to completion of their response without being preempted by higher priority tasks. To account for preemptions we had to recognize them in the trace. The mere fact that a higher priority task T_{high} is executed right after a lower priority task T_{low} does not indicate that the lower priority task was preempted. It is possible that the lower priority task T_{low} had completed its response before T_{high} was scheduled. To recognize the preemption, our system produced a trace record at the end of each task's response.

Unfortunately, we could not identify ways to discover deadlines from analysis of execution traces. Some of the deadlines are due to protocols of interaction with other systems and thus cannot be determined from traces in principle. In this study, we had interviews with software designers in order to elicit the deadline information.

Once we have extracted from the traces the observations of each task's parameters such as invocations and execution times, we looked at the obtained data from the scheduling point of view. Immediately we noticed a number of issues that are not handled by the fixed priority scheduling methods.

In simple scheduling approaches, each task is characterized by a single period, deadline, and execution time. Of course, it is commonly understood that in many, possibly in most, cases the tasks are not perfectly periodic. A typical recommendation from scheduling literature, however, is to look at the worst case. This means using the shortest period and the longest execution time. Unfortunately, in all the use cases we have analyzed, such a recommendation is not useful because none of these use cases would be schedulable even though we knew that the system performed fine in practice. For example, in a simple voice call scenario, where the average processor load was lower than 30% and the system was clearly schedulable, the worst-case execution time in a single task was larger than its worst-case period, which indicates that even this single task cannot be scheduled. Similarly, Table 1 shows the extracted data of a communication task for the use case of file transfer between a laptop and a server over USB and GPRS. The worst-case period of the task takes 0.4 time units, while the worst-case execution time takes 20 time units.

Table 1. Communication task execution times and periods in file transfer over USB and GPRS

Worst case (longest) execution time	20	Worst case (shortest) period	0.4
Average execution time	1.6	Average period	20.5
Largest cluster execution time	1.3	Largest cluster period	1.6
Execution time clusters		Period clusters	
Time	Size	Time	Size
1.3	4352	1.6	1479
3.1	906	4.3	657
		10.6	562
		20.6	1024
		41.9	1221
		100.9	280

4. MULTIVARIATE TASKS

The assumption that tasks have a single period and execution time or that they are single peak statistical functions does not hold in our data. However, it may be possible to identify a small set of typical ("peak") periods and execution times of a task. We called this a multivariate task hypothesis.

To verify this hypothesis we developed a data-clustering algorithm that identifies the presence of multiple clusters ("peaks") in observations of task periods and execution times. We have applied a modified K-means algorithm with criteria for establishing new clusters and merging not sufficiently separated clusters. The results of cluster analysis confirmed the multivariate task hypothesis—most of the tasks had several characteristic periods and execution times. For example, a communication task (Table 1) has two execution time clusters and six period clusters in the given use case. Most of our real-time tasks could be characterized by a small set of parameter vectors. These data also indicated that such statistical measures as average period and execution time are not representative of the mobile device's task behavior. Indeed, although the average execution time divided by the average period represents the average processor utilization contributed by a task, this load does not represent a reliable schedulability measure (also noted in [4]). Since tasks have different characteristic periods and execution times, the average may not represent any single cluster. For example, the average execution time of the task in Table 1 does not correspond to either of the two large execution time clusters. This observation is typical for other tasks as well.

We analyzed the clusters further. In many cases, it is impossible to perform the schedulability analysis based on just the largest execution-time and period clusters. For example, the largest period and execution-time clusters of the task in Table 1 show that this task would have utilized the processor at $1.3/1.6 \approx 81\%$ utilization. Since the system has other tasks as well, such a high utilization by one task indicates that the system is unschedulable. However, as we know from experimental data, this is not the case. So the schedulability analysis needs to take into account the relationship between different execution time and period clusters. With such understanding it may be possible to identify pairs of corresponding period and execution time values. It is also essential to determine which parameter values of one task correspond to which parameter values of another task from the same featureset.

What causes multiple period and execution time clusters in a single task? If tasks were designed with runtime architecture and schedulability in mind, a single task should be assigned only objects handling a single event stream [1][6]. However, in real systems, tasks often contain objects handling more than one stream of events. These event streams may be independent and have different periods and execution times. The observed periods and execution times will be a complex superposition of periods and execution times of different event streams and may not display any characteristic period at all. In depth understanding of tasks may allow to separate the multiple event streams and determine periods and execution times characterizing each of them. We found that this is difficult to achieve from the traces and usually requires domain expert help.

Multivariate task scheduling is not fully addressed by fixed priority scheduling methods, requiring new extensions.

5. ACTIVITIES

In our system a response to an event may involve multiple tasks of different priority. The invocation of these tasks is deterministic in response to an event and thus has to be seen as a single activity. Furthermore, some responses may have multiple deadlines for different tasks that constitute the response.

We call multitask responses to events *activities*. An *activity* may be represented as a message sequence chart ("sequence diagram") of task executions connected by messages sent from one task to other tasks. An activity starts with an external event, for example, a timer expiration, an interrupt, or another similar event. An activity ends when all tasks involved in the activity are finished with their responses. In some scheduling research [2][5], parts of activities are called *subtasks* and the activities themselves are called *tasks*. In our context, such naming leads to confusion, since activities are not operating system level tasks while "subtasks" are really operating system level tasks. Therefore we use the terms activities and tasks.

It is evident from extracted activities that phone tasks are highly dependent on each other. An external event is handled not by just one task, but by a number of communicating tasks. This means that whole activities need to be considered in real-time analysis as well as in scheduling. A single priority cannot be assigned for an activity composed from multiple tasks. We present a few alternative ways of dealing with this.

5.1 Priority assignment for tasks in activities

First approach is to analyze activities and to assign task priorities for tasks in activities. Klein et al. [4] and Harbour et al. [2][3] show how to determine the system schedulability for systems with activities. However, we did not find any results that show an optimal priority assignment for systems with activities.

Harbour et al. [3] suggest a heuristic of assigning priorities to tasks in activities according to a deadline monotonic algorithm. However, this approach is proved optimal only for a limited set of schedules where all tasks in activities have nonascending priorities. On the other hand, Harbour et al. [2][3] use the activity's canonical form, which is obtained by converting all task priorities to a nondescending form. The conversion is done by starting from the end of an activity and lowering priorities of any tasks that are higher than later task priorities (detailed algorithm is given in [2]). It is shown that the activity completion time is the same for original and canonical forms. This seems to argue for the use of the nondescending priority assignment for tasks in an activity. However, no proof is given that such assignment is "good". We provide several new conjectures regarding such assignment below.

Conjecture 1. Consider the simplest situation where there are no internal deadlines for the tasks in an activity A and no tasks are shared between activities. Conversion of activity A to a canonical form improves schedulability of the system.

Proof. Harbour et al. proved that the activity A completion time is the same for original and canonical forms. Since all tasks in activity A after the conversion to a canonical form have lower or the same priority than before conversion (see conversion algorithm in [2]), they can interrupt or block fewer other activities and tasks than before. Therefore other tasks and activities have the same or shorter interruptions or blocking times and finish at the same time or earlier improving their schedulability. QED.

Conjecture 1 shows that for every schedule in a system with above constraints there is a better or equal schedule in which all activities are in canonical forms. Such a schedule can be constructed by converting each activity in turn into a canonical form. Therefore the optimal schedule for a system with no intermediate deadlines in activities and no tasks shared between activities is a schedule with all activities in a canonical form.

Unfortunately this result does not hold anymore if the constraints on the system are changed. If tasks in activities have internal deadlines, these deadlines can be broken by the activity conversion to a canonical form. Consider Figure 3. It shows two activities: one consisting of task 1 and task 3 and another one consisting of task 2. Task numbers are shown inside task execution bars. Task 1 has execution time $C_1 = 2$ and deadline $D_1 = 2$. Execution times of other two tasks are $C_2=4$, $C_3=2$. Periods of activities are $T_{A1} = 13$, $T_{A2} = 7$. Task 1 satisfies internal deadline of two time units in the original priority assignment, where it has the highest priority of all tasks. However, task 1 fails the deadline when the activity is converted to canonical form and task 1 priority is lowered to the priority of task 3.

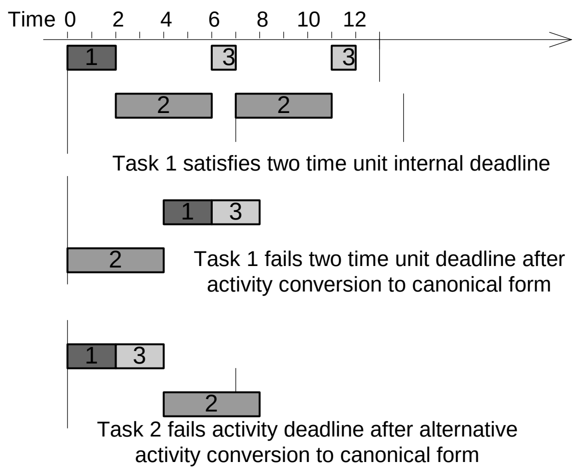
Time 0 2 4 6 8 10 12
1
3
3
2
2
Task 1 satisfies two time unit internal deadline
1
3
2
Task 1 fails two time unit deadline after activity conversion to canonical form
1
3
2
Task 2 fails activity deadline after alternative activity conversion to canonical form

Figure 3. Conversion of activity with internal deadlines to canonical form

Is there an optimal priority assignment in this example such that all activities are in canonical form? No. P_1 (priority of task 1) has to be greater than P_2 for task 1 to satisfy its internal deadline D_1. Which means that $P_3 \geq P_1$ (by canonical form) $> P_2$. But this is the situation at the bottom of the Figure 3, where task 2 fails the activity deadline $T_{A2}=7$. On the other hand, this task set with a non-canonical form schedule is schedulable as shown in the top of the Figure 3.

Conjecture 2. If tasks have internal deadlines, the optimal schedule does not always have all activities in canonical form.

If tasks are shared between activities, but no tasks have internal deadlines, the usefulness of conversion to a canonical form depends on whether same priority tasks are allowed and how same priority tasks are scheduled.

Consider Figure 4. In it $C_1=2$, $C_2=2$, $C_3=1$, $T_{A1}=6$, $T_{A2}=12$. Task 2 is shared by both activities. In original priority assignment $P_3<P_1<P_2$. Activity 1 is in canonical form, while activity 2 is not in canonical form. The system is schedulable. If activity 2 is converted to a canonical form, P_2 becomes equal to P_3. However, now activity 1 is not in canonical form: $P_2=P_3<P_1$. Now activity 1 is converted into a canonical form and priorities become $P_1=P_2=P_3$. However, the scheduling of such a system depends totally on the operating system scheduler implementation and in the "bad" case (Figure 4 bottom) activity 1 fails its deadline.

Therefore, if activities have shared tasks, the conversion to the canonical form should be used only if scheduling of tasks with the same priority and scheduling of the same task invoked from different activities is well understood. Systems usually are not designed to have tasks with the same priorities because then the internal scheduler implementation determines the processing order of same priority tasks. For example, if two tasks of the same priority become ready, the scheduler has to decide which one will be executed first.

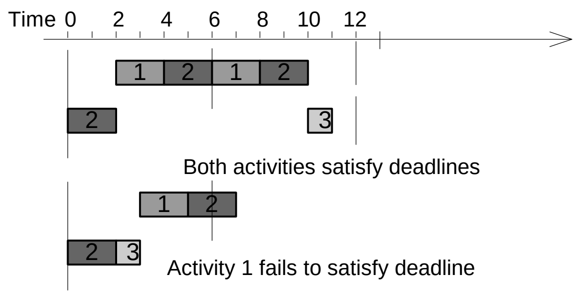

Figure 4. Activities with shared tasks

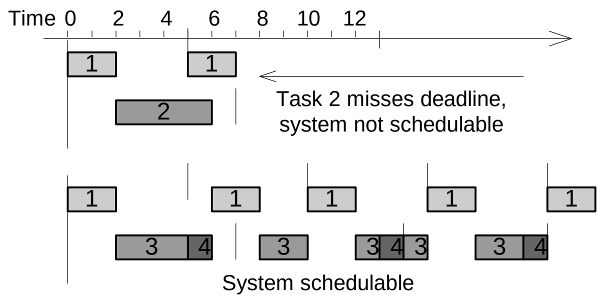

Figure 5. Harbour et al. example

5.2 Task merging and splitting in activities

The second approach to activity scheduling is to simplify the system first by moving all the functionality performed in an activity into a single task. This allows assigning a single priority to the activity and using known scheduling algorithms for the priority assignment. However, such reallocation is not possible if the same task's actions are needed in different activities. Reallocation is also impossible if tasks in an activity have internal real-time constraints. Finally, reallocation leads to task merging, which, as observed in [2], reduces overall system schedulability. To repeat Harbour et al. example (Figure 5): tasks 1 and 2 are not schedulable using optimal RMA priority assignment ($P_2<P_1$), but are schedulable if task 2 is split into tasks 3 and 4, where task 3 keeps task's 2 priority, but task 4 has the highest priority ($P_3<P_1<P_4$). Details of this example are covered in [2].

We look at this example from the opposite direction. If system has tasks 1, 3, and 4 in the beginning, the system is schedulable, but it is no longer schedulable after merging of tasks 3 and 4 that constituted an activity. We suggest the following new conjecture.

Conjecture 3. Splitting activity into tasks increases system schedulability. Merging tasks decreases system schedulability.

Proof. The conjecture is intuitively obvious, since task splitting increases degrees of freedom and merging decreases degrees of freedom. Consider two systems such that some task T is split into tasks T1, …, Tn in the second system. The second system can always be scheduled in the same way as the first system, by assigning tasks T1, …, Tn the same priority P which was held by task T. If the system does not allow equal priorities, assigning priority series P+δ, …, P+nδ, such that no other priority P' falls into interval [P, P+nδ] assures the same schedule for the second system as for the first system. This shows that the system with split tasks is always at least as schedulable as the system from which the split was done. QED.

The above proof assumes that there is no task-switching overhead. Task splitting increases the number of task switches, which means that the overhead due to task switching increases too. This may become an issue if the task-switching overhead is large.

Harbour et al. [3] proved that any system of two tasks with deadlines equal to periods and system utilization ≤ 1 is schedulable by splitting the longer period task into two and assigning appropriate priorities. As far as we know, there is no similar proof for an arbitrary number of tasks. For task sets with deadlines shorter than periods, Harbour et al. [3] result does not hold even for two tasks. Consider the system in Figure 5. If we make task 1 deadline D_1 equal to its execution time C_1, task splitting cannot make the system schedulable anymore. If task 2 is not split, it misses its deadline. If task 2 is split, task 1 misses its deadline.

It seems that people scheduling a system with activities are left in a quandary: they can stay with activities, but not know the optimal priority assignment, or they can merge tasks decreasing the system schedulability. In our project, we could not simply merge activity tasks, since tasks were invoked multiple times in an activity and they participated in multiple activities.

6. CONCLUSIONS

Our project work on runtime architecture of mobile phone software led us into the systematic scheduling analysis of mobile phone tasks. We believe that the issues raised, extensions suggested and future directions outlined can bring the fixed priority scheduling methods to more real-world industrial projects and contribute to good systematic scheduling practices.

7. REFERENCES

[1] H. Gomaa, *"Designing Concurrent, Distributed, and Real-Time Applications with UML"*, Addison-Wesley, 2000.

[2] M.G. Harbour, M.H. Klein, J.P. Lehoczky, "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 116-128, Los Alamitos, CA: IEEE Computer Society Press, 1991.

[3] M.G. Harbour, M.H. Klein, J.P. Lehoczky, "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 13-28, IEEE Computer Society Press, 1994.

[4] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, *"A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems"*, Kluwer Academic Publishers, 1993.

[5] J.W.S. Liu, *"Real-Time Systems"*, Prentice-Hall, 2000.

[6] A. Ran, R. Lencevicius, "Making Sense of Runtime Architecture for Mobile Phone Software", *Proceedings of ESEC/FSE'2003*.

[7] K.W. Tindell, A. Burns, A.J. Wellings, "Mode changes in priority preemptively scheduled systems", *Proceedings of the Real-Time Systems Symposium 1992*, pp. 100-109, 1992.