A Tree-Driven Multiple-Rate Model of Time Measuring in Object-Oriented Real-Time Systems

Vesna Minčev[1] and Dragan Milićev[2]

[1]*The Institute for Telecommunications and Electronics IRITEL, Belgrade*

[2]*University of Belgrade*

Abstract – This paper addresses the problem of measuring time in object-oriented event-driven real-time systems. Several possible implementations of the timing subsystem are considered. Some important consequences that an implementation can have on the overall time overhead are discussed.

The paper proposes a modified multiple-rate policy of updating software timers. Timing ticks are distributed through a tree structure in a way that tends to minimize the overall overhead, while retaining a small variation of the overhead in different clock cycles.

The paper presents the results of a comparative analysis of the described models. The analysis considers different exploitation conditions, in respect to the number of software timers and their resolutions.

Keywords: object-oriented real-time systems, event-driven systems, time measuring

1 Introduction

Measuring time is among the very fundamental functions to be realized in real-time systems. Working on software development for the digital telephone switching system DKTS 30, we have faced the necessity to have an accurate, compact, and flexible subsystem for time measuring. Being a part of the software kernel for the DKTS 30 system, it had to be also reliable, reusable, and efficient, meaning that its time overhead had to be minimized.

In order to achieve reusability, we had to build a conceptually clear object-oriented time-measuring subsystem with a simple interface to other parts of software [1,2,4]. Such an approach compelled us to study carefully different implementations in order to obtain best performance. We have compared several possible solutions and proposed a new one that we call the Tree-driven Multiple-rate model. This paper describes the proposed model, presents the results of our analysis and the conclusions of the comparison.

The paper continues as follows. Section 2 states the problem more precisely. Section 3 describes two existing solutions that we compared. Our new model is presented in Section 4. Conditions of our comparative analysis and its aims are described in Section 5. The experimental results are shown in Section 6. The paper ends with conclusions.

2 Problem Statement

Since the object-oriented software for the DKTS 30 system was developed in C++, built-in language concepts for time measuring were not available. Besides, we wanted to obtain platform-independent software. Thus, we did not use directly timing

primitives of an existing operating system [9,3,7,11]. Finally, we were not in a possession of a ready-to-use C++ library that we could easily tune for performance. That is why we have developed our original time-measuring subsystem that has a simple interface and different possible implementations even in different application software subsystems of the same system.

In an event-driven real-time system [8] such as a telephone exchange, there is a need to measure time intervals in two different manners: (a) to start measuring time when an event occurs, and to stop it when another event occurs, i.e., to measure time intervals of unknown duration, and (b) to start measuring a known time interval when an event occurs, and to take an action if the time is out, i.e., to measure time-out intervals [3,7,11].

Another major problem was that we expected a wide range of time intervals to be measured–from 10 ms up to a few hours. In order to minimize the overhead, we had to introduce the concept of resolution: shorter time intervals should be measured with higher resolution, while for the longer ones a smaller resolution would satisfy. As a result, we have built an object-oriented timing subsystem, with an abstraction of Timer. A Timer object is a software device that has the following interface in C++:

```
class Timer {
public:
  Timer (Timeable* toWakeUp);

  void start (Time interval, Time resolution);
  void stop  ();
  Time elapsed () const;
};
```

A Timer object is associated with a Timeable object to which the Timer sends a timeout message when the given time interval is out. Timeable is an abstract base class with the interface that accepts the timeout message. Concrete derived classes provide the needed behavior for a timeout. A Timer object is assigned a resolution upon startup. It can be started, stopped, or asked for the elapsed time by the corresponding operations. This concept has proved as a simple yet powerful mechanism for concurrent time measuring of a great number of intervals.

A single hardware interrupt, which is the impetus of the subsystem, is generated every 10 ms and it gives the basic rhythm. The software timing subsystem has the responsibility to update all the existing Timer objects, i.e., to distribute a tick to the timers. As this process is time-consuming, but also time-critical, it should be carefully tuned for performance. First, there is no need to update all the timers on each hardware tick: the smaller the resolution, the less often a timer should be updated. Second, the internal organization of lists of timers has a significant impact on updating iteration overhead. Besides, the updating overhead should be nearly equal at each hardware timer's tick.

Consequently, our problem was to find out an implementation with the smallest overall overhead in different exploitation conditions, as well as with an equally distributed overhead over time. Under 'different conditions' we assume different distributions of the number of activated timers in respect to the existing resolutions. It can be expected that different implementations of the updating mechanism behave differently when these distributions vary.

3 A Survey of the Competitive Solutions

A straightforward solution to the problem is to organize the timing subsystem as in Figure 1. The TimerController object receives hardware timer's ticks. A TimerManager object is responsible for updating Timer objects of a single resolution (R_i denotes a resolution). The TimerController object forwards a tick to all the TimerManager objects at each hardware tick, denoted in the figure with tick[1]. A TimerManager R_i object forwards a tick to its timers only at each R_i-th received tick.

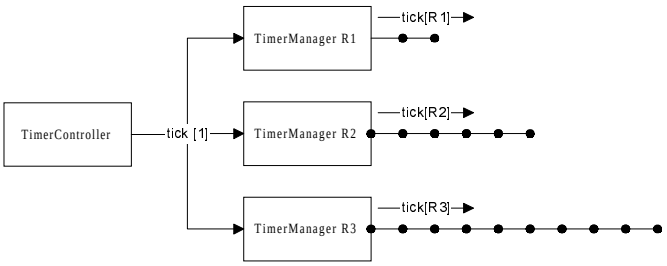

Fig. 1. A simple solution to the problem

The advantages of this approach are its simple implementation and a small overhead for updating lists of timers. Its obvious drawback is that the overhead dramatically differs in various ticks: at each LCM(R_i, R_j)-th tick all the timers of resolutions R_i and R_j should be updated. That is why we do not consider this approach further.

Another approach is used in the previous generation of our DKTS system. Its idea is shown in Figure 2. The TimerController object distributes a tick to all the TimerManager objects at each hardware tick. A TimerManager R_i distributes a tick to all its timers at each R_i-th received tick, but the moments of distribution of the ticks to the timers are skewed between the TimerManager objects. For example, the timers with the resolution 10 tick at 1, 11, 21, 31, etc., while the timers of resolution 50 tick at 2, 52, 102, 152, etc.

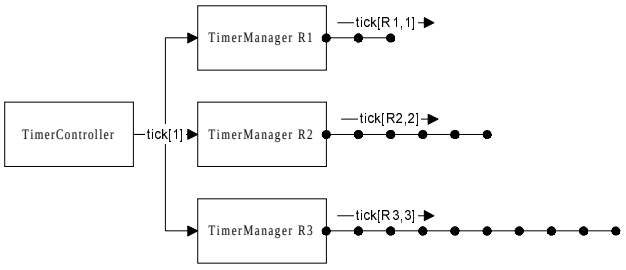

Fig. 2. The Model 1

This approach does not suffer from the large overhead at LCM(R_i, R_j)-th ticks due to the tick distribution skew. This is why it can be expected that this solution behaves well in exploitation. Nevertheless, this model has a drawback that all the timers of one resolution are still updated at the same tick. When the number of timers

of different resolutions differ significantly, which can be expected, this model still has an unequalled distribution of the overhead over different ticks. For example, there are much more timers of resolution 10 and 50 time units in our DKTS system then all the others. Therefore, each 10^{th} and 50^{th} tick has a great updating overhead, as opposed to the others. We will further refer to this approach as the Model 1.

Finally, we will briefly describe another model, inspired by the task scheduling policy [5] known as the Multiple-rate Table-driven approach. In this model, which we will further refer to as the Model 2, the timers of the resolution R_i are organized in R_i lists with nearly equal number of timers (Figure 3). Each TimerManager object updates the timers of one of its lists at each received tick.

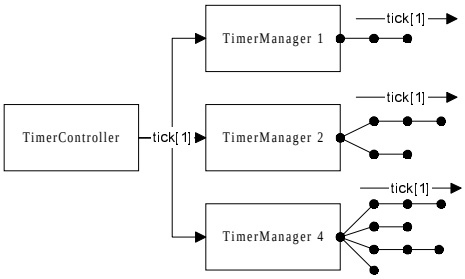

Fig. 3. The Model 2

This solution tends to equalize the overhead at each hardware tick, independently on the number of timers of each resolution. This can be obtained by attaching newly started timers of the resolution R_i to the TimerManager's lists in round-robin policy, i.e., according to their sequence of creation. However, this policy may reduce the accuracy of time measuring: a newly started timer may be attached to the list that is to get a tick at the very next moment. This may lead to a loss of accuracy because the timer may stop counting R_i time units before it is really expected. If R_i is large (a small resolution), the loss may be significant.

In order to improve accuracy, we have adopted another policy. When a timer of resolution R_i is started, it is attached to the list of the TimerManager R_i which has been just updated in the previous tick. Consequently, the timer is going to get the next tick in approximately R_i time units. This improves the accuracy, while still keeping nearly equal distribution of the timers in the lists: if the timers are started at random moments, they will be randomly attached to the lists, and the lists will be of nearly equal size on average.

On the other side, the disadvantage of this model is that if we have to measure long time intervals with large R_is (small resolutions), we will have a great number of lists in each TimerManager object. If the total number of timers of the resolution R_i is relatively small, the lists will be almost empty. This can unnecessarily introduce an extra overhead in a TimerManager object while setting up an iteration through a list at each tick, i.e., the utilization can be low.

4 The Proposed Model

Unlike the previous two models in which every tick is propagated to each TimerManager object, we propose a model in which the TimerController is attached to a single TimerManager object, and all the TimerManager objects R_1, R_2, ..., R_i, ..., R_n are linked in a structure as in Figure 4. For the moment, it is assumed that each R_{i+1} is an integer multiple of R_i. The number of lists in the TimerManager R_i is $L_i = R_i/R_{i-1}$. Each TimerManager R_i propagates only each L_i-th received tick to the successive TimerManager R_{i+1}. When a TimerManager gets a tick, it distributes it to one of its lists, circularly. Consequently, each timer of the resolution R_i receives a tick every $L_i \cdot L_{i-1} \cdot \ldots \cdot L_1 = (R_i/R_{i-1}) \cdot (R_{i-1}/R_{i-2}) \cdot \ldots \cdot (R_2/R_1) = R_i/R_1 = R_i$ hardware ticks, where $R_1 = 1$, which was intended.

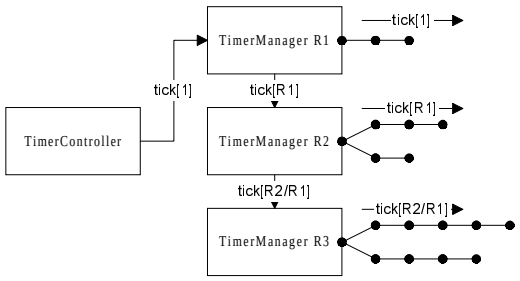

Fig. 4. The basic idea of the proposed model

For the example in Figure 4, the TimerController sends each tick to the TimerManager R_1 only (R_1=1). The TimerManager R_1 has only one list and updates all its timers at each received tick. It also propagates each received tick to the next TimerManager R_2 (R_2=2). This one has two lists and updates only one list of timers at each received tick. However, it propagates each 2^{nd} received tick to the next TimerManager R_3 (R_3=4), etc. Thus, the TimerManager R_3 receives a tick every 2 time units, and each of its timers receives a tick every 4 time units.

In a general case, when R_{i+1} is not a multiple of R_i, it is possible that a TimerManager has several successive TimerManagers. Consequently, we can have a tree topology of TimerManagers structure, as shown in Figure 5. We will refer to this model as the Tree-driven Multiple rate strategy, or the Model 3.

This approach has not a strictly equalized distribution of overhead on each tick. It sacrifices a part of this property for a better expected utilization in the case when there is a relatively small number of timers of a large R_i, since there is a relatively small number of lists to be iterated through and the iteration overhead is therefore reduced. It can be expected that under some distributions of the number of timers over the resolutions, this approach will lead to a better overall performance than the other models. In real-life systems, it can be easily expected that the distribution of timers really has the property that some resolutions are dominant over the others.

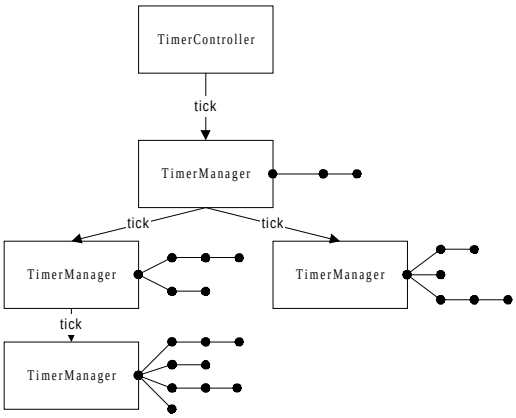

Fig. 5. A general structure of the tree-driven strategy – Model 3

5 The Comparative Analysis

Before we describe our analysis strategy, we would like to emphasize that our aim was to assess each of the proposed models from the practical point of view. We needed a simple and fast analysis that could still outline some useful conclusions. Therefore, we decided to conduct the experiment in which we would evaluate some basic parameters of the model quality, justifying this way the selection of a particular model [6].

As we have already explained, our objective was not only to minimize the overall time overhead in a time unit (a time unit is one clock cycle, i.e., the period between two hardware timer's ticks), but also to equally distribute it over time units. Consequently, the mean value and the standard deviation of the time overhead per time unit in our case appeared to be the parameters of considerable importance. As the 'overhead' we consider the time needed to iterate through the structure and to update the corresponding timers in each time interval.

The input parameters domain included:

1. the set of values of resolutions;
2. the number of different resolutions,
3. the total number of activated timers;
4. the distribution of timers in respect to the resolutions.

We have set the second and the third parameters as constants and assigned them the values in regard to our empirical estimations, in order to achieve a tractable model. Accordingly, we de facto tested the efficiency of the proposed models by measuring the mean value and the standard deviation of the time overhead per time unit, in respect to the specified set of resolutions and the distribution of the number of timers among TimerManagers.

The tests were conducted for the total number of timers 1008. This particular value was chosen because it was suitable for testing the uniform distribution (divisible by 7 – we had seven values in each resolution set) and because we estimated that in practice we would unlikely have more than a few hundred timers.

Three groups of tests were conducted, each group for one of the following sets of resolutions:

1. {1, 5, 10, 50, 100, 500, 1000}
2. {1, 4, 8, 44, 96, 480, 1056}
3. {1, 4, 9, 49, 99, 499, 999}

The first set of values is the one currently used in our DKTS system. As it can be easily noticed, each value in the set is divisible by the previous value, and therefore, instead of having the tree topology of the Model 3, we have a linear structure. Accordingly, in order to examine the advantages and disadvantages of the proposed models in the general case, we have chosen the other two sets that contain values approximate to those in the first set, but that enforce the tree topology of the Model 3. The third set represents the specific case in which the tree structure of the Model 3 contains only a root and leaves, i.e. there are no intermediate nodes.

Each group contained four tests, one for each chosen distribution. Figures 6 to 9 represent the chosen distributions. The particular values of the number of timers for each resolution for the tests 1 and 2 were adopted using the formula for the standard Gauss distribution for the range [0,6] with the mean value equal to 0 and the standard deviation equal to 3. The values for the test 3 were chosen in a similar way, only the mean value was 3 and the standard deviation was 1.5. Apparently from the Figure 9, the test 4 was conducted for the uniform distribution.

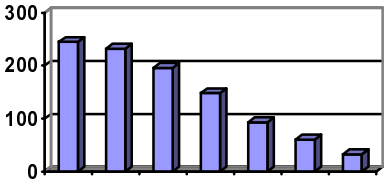

Fig. 6. The test 1 distribution

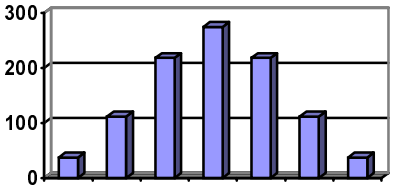

Fig. 8. The test 3 distribution

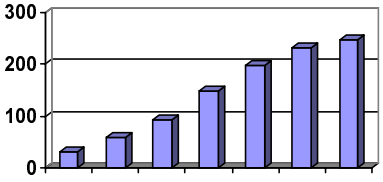

Fig. 7. The test 2 distribution

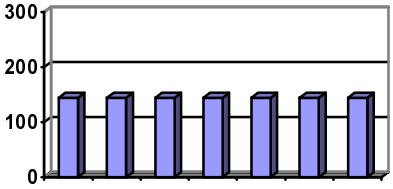

Fig. 9. The test 4 distribution

As the criterion of the model efficiency we used the average time overhead in a time unit and its standard deviation for a chosen testing time interval. Achieved values were relative indicators of the models' efficiency.

6 The Results of the Analysis

The average time overhead and the standard deviation of the time overhead per time unit for the tests 1 to 4 (different distributions of the number of timers) are shown

in figures 10 to 13, respectively. Each figure shows the results for the three groups of tests (different resolution sets) and for the three compared models.

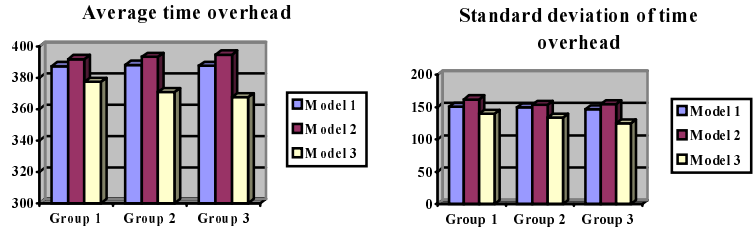

Fig. 10. Test 1 results

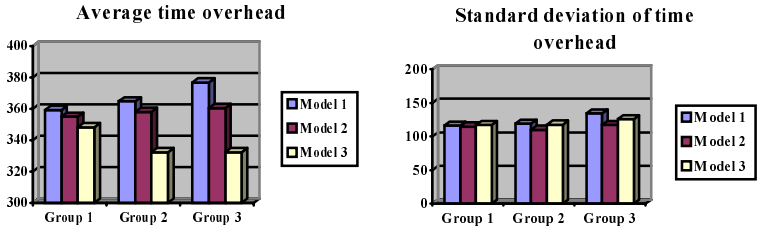

Fig. 11. Test 2 results

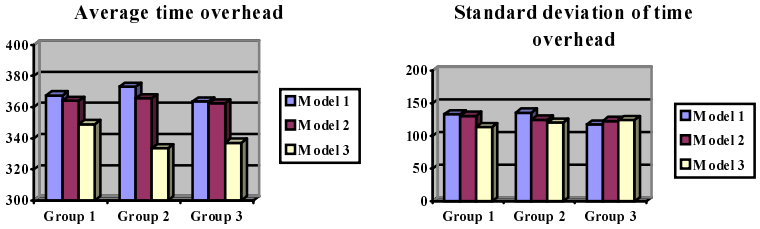

Fig. 12. Test 3 results

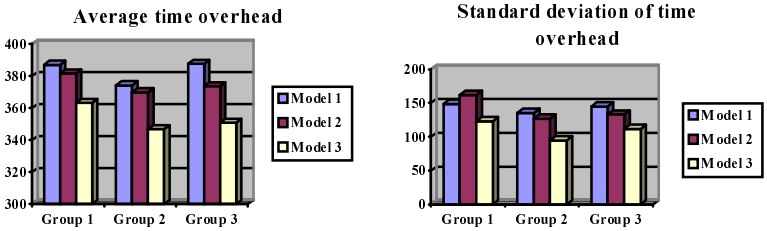

Fig. 13. Test 4 results

Our expectation that the models would show the smallest differences for the test 1 distribution was confirmed for each of the three groups of tests. As it can be easily noticed in Figure 10, the differences between the mean values of the time overhead as well as between the standard deviations are not very large, but the model 3 appeared to have a slightly better performance though.

Major differences were expected for the test 2 and 3 distributions because there were more timers with smaller resolutions (larger R_is). As it can be seen in figures 11 and 12, the average time overhead appeared to be considerably reduced in the model 3 compared to the models 1 and 2, while its standard deviation remained approximately the same. The meaning of these facts is that the total time overhead, as well as the time overhead in a particular time unit, has been reduced.

The test 4 distribution was the very specific case with a rather small probability to appear in practice, but interesting from the theoretical point of view.

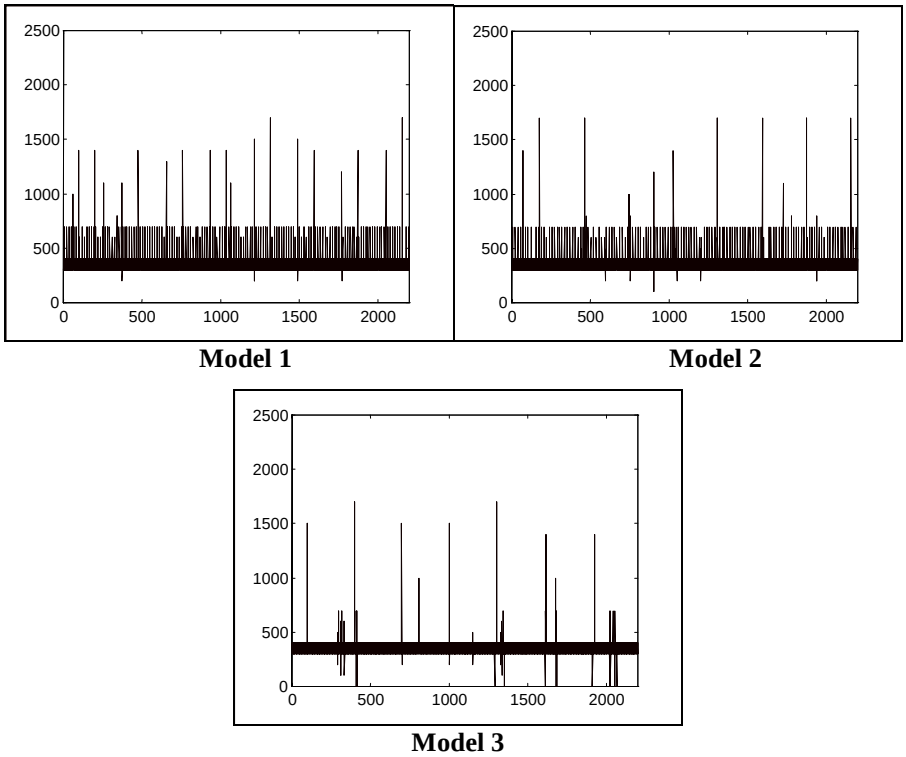

Fig. 14. Time overhead duration in one of the tested cases

In order to illustrate the meaning of the output parameters, we have enclosed three diagrams in Figure 14 that represent the time overhead during the testing time interval of 2200 time units. The diagrams correspond to the group 2 testing for the test 4 distribution of timers, so that their characteristic statistical values are shown in Figure 13. According to these values, the model 1 has the largest average time overhead and the largest standard deviation. Translated into the following diagrams terms, it means that the time overhead varies more for the model 1 than for the model 2. The difference is greater in respect to the model 3, which is also noticeable from the diagrams.

It is fair to say that the proposed solution is a probabilistic one and, as Figure 14 shows, the time overhead considerably varies in all models. Moreover, the model 3 does not actually improve the worst-case behavior, which is relevant for hard real-time systems. Therefore, the suggested approach is applicable only for soft real-time

systems. However, our target DKTS 30 system is a soft real-time system and we intend to incorporate the proposed solution in it.

7 Conclusions

We have addressed the problem of measuring time in object-oriented event-driven real-time systems. We have described several possible implementations of the timing subsystem, and discussed some possible consequences that the implementation can have on the time overhead.

We have proposed a modified multiple-rate policy of updating software timers. Timing ticks are distributed through a tree structure in a way that tends to minimize the overall overhead, retaining a small deviation of the overhead in respect to different time units.

We have conducted a comparative analysis of the described models that has proved the expectations. Our proposed model has a smaller overall overhead than the other considered models, with the comparative or even smaller deviation in all the considered cases.

As a conclusion, we intend to incorporate the proposed model in our new DKTS telephone switching system, and expect to use the results and conclusions of our analysis for further theoretical and practical studies of the subject.

References

[1] Allworth, S. T., Zobel, R. N., *Introduction to Real-Time Software Design*, Second ed., Springer-Verlag, 1987

[2] Booch, G., *Object-Oriented Analysis and Design*, Second ed., Benjamin-Cummings, 1994

[3] Furht, B., Grostick, D., Gluch, D., Rabbat, G., Parker, J., McRoberts, M., *Real-Time Unix Systems Design and Application Guide*, Kluwer Academic Publishers, 1991

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley, 1995

[5] Laplante, P. A., *Real-Time Systems Design and Analysis*, Second ed., IEEE Computer Society Press, 1997

[6] Pflegeer, S.L., Jeffery, R., Curtis, B., Kitchenham, B., "Status report on Software Measurement," *IEEE Software*, Vol. 14, No. 2, pp. 33-43, 1997

[7] *pSOSystem System concepts*, Integrated systems Inc., 1994

[8] Selic, B., Gullekson, G., Ward, P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, Inc., 1994

[9] Silberschatz, A., Peterson, J. L., Glavin, P. G., *Operating Systems Concepts*, Third ed., Addison-Wesley, 1994

[10] Stankovic, J., Ramamritham, K., *Hard Real-Time Systems–A Tutorial*, IEEE Computer Society Press, 1988

[11] *System Software for M7-300/400 Program Design, Programming Manual*, Siemens AG, 1997