



University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Simulation-Based RMA

Jaroslav Kačer, Stanislav Racek

Technical Report No. DCSE/TR-2005-01
February, 2005
Version: 1

Distribution: Public

Technical Report No. DCSE/TR-2005-01
February 2005

Simulation-Based RMA

Jaroslav Kačer, Stanislav Racek

Abstract

This paper describes the very first version of the UWB/Apogee RMA Toolkit. The goal of the toolkit is to provide a simulation-based way to verify schedulability properties of real-time Java programs. The initial version described in this paper – version 0.1.0 – is a purely theory-based solution with no direct relation to Real-Time Java. Properties of a real-time program are described by means of its tasks' characteristics, such as period length, deadline, cost, and (optionally) priority, and by means of the system's properties, such as the number of available priority levels and the preemption latency. The output of a simulation execution is a sequence of decisions assigning processor time to tasks and reports whether tasks complete before their deadlines or not.

Copies of this report are available on
<http://www.kiv.zcu.cz/publications/>
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Copyright ©2005 University of West Bohemia in Pilsen, Czech Republic

Contents

1	Introduction	2
2	UWB/Apogee RMA Toolkit	4
2.1	Fundamental Classes	4
2.1.1	Task	4
2.1.2	Scheduler	6
2.2	Naming Conventions	7
2.3	Priority Assignment	8
2.3.1	Manual Priority Assignment	9
2.3.2	Automatic Priority Assignment	9
2.4	Execution	11
2.4.1	Complexity	13
3	Future Extensions	14
3.1	Respecting Variations of Task Execution Time	14
3.2	Handling Task Synchronization	14
3.3	Aperiodic Tasks	15
3.4	Execution Time Detection	15
3.5	GUI	16
4	Conclusion	17

1 Introduction

The Rate Monotonic Analysis is a method of assigning fixed priorities to periodic tasks of a real-time program to maximize the program's schedulability. Every task of the program is given a period T_i and a worst-case execution time $C_i, C_i < T_i$. The program is schedulable if all tasks meet their deadlines, i.e. if they *complete all work before their deadlines* which are by default at the end of their periods.

The Rate Monotonic Analysis was first developed at the Software Engineering Institute of Carnegie Mellon University as a part of the RMARTS project. Its brief description can be found in [RMA].

The RMA assigns each task its priority according to its period so that the shorter the period the higher the priority.

The basic schedulability test¹ says that a program is schedulable if the following condition is met:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq U(n) = n(2^{\frac{1}{n}} - 1)$$

Here n is the number of tasks in the system. $U(n)$ is called the *processor utilization bound* and its limit goes to $\ln 2$, i.e. about 0.693, if n goes to infinity. Basically, there are three possibilities:

1. If $\sum \frac{C_i}{T_i}$ is less than $U(n)$, the program is schedulable.
2. If $\sum \frac{C_i}{T_i}$ is greater than 1, the program is not schedulable.
3. If $\sum \frac{C_i}{T_i}$ is greater than $U(n)$ and less than 1, this basic test is inconclusive and additional tests must be performed in order to decide about the program's schedulability. An example of such tests is the *Response Time Test*.

The basic RMA theory was later extended with other topics like aperiodic tasks and task synchronization.

Aperiodic tasks are handled by means of an aperiodic server which has a certain execution budget and a replenishment period. As long as the

¹Also called *The Utilization Bound Test*.

execution budget is available, the aperiodic server handles coming requests at their desired priority. As soon as the budget is spent, incoming requests are handled at a low priority until the budget is replenished. There are three basic types of aperiodic server, distinguished mainly by the algorithm of budget replenishment.

Task synchronization brings other problems. First, additional time, needed for synchronization, must be taken into account when doing schedulability tests. When semaphores, a common synchronization technique, are used, a lower priority task can block a higher-priority task for an unbounded time period. This phenomenon is called *unbounded priority inversion* and it can be solved by application of the priority ceiling protocol that adjusts the blocking task's priority if a higher-priority task is waiting until the blocking task exits a critical section.

This work focuses on the case when the basic schedulability test is unable to decide about a system's schedulability. (But it can be used in any case, of course!) Although other approaches are possible, such as the Scheduling Point Test [Dibble, 91-94], we found the simulation approach convenient because of possible future extensions. The Scheduling Point Test is a purely mathematical method to verify whether a system is schedulable or not. Compared with the present version of the RMA Toolkit, the results are exactly the same, just the method differs. We believe that the simulation approach gives us free maneuver space if tasks are (in the future) described in a more complicated way than just by a period and a cost. For example, a task can be described as a sequence of computing actions and interactions with other tasks. Purely theoretical methods would probably fail here because they are not able to enumerate the exact impact of mutual task synchronization. (The Scheduling Point Test can be extended with synchronization, but the maximum times spent for synchronization must be known.) Also, implementing aperiodic tasks seems to be a fairly simple job when the simulation approach is used.

The RMA Toolkit is written in Java. It has a form of a Java library. We assume that the user will create a simple program using the library and execute it. Two examples are provided in version 0.1.0 that can be used as a template for experiments.

2 UWB/Apogee RMA Toolkit

The UWB/Apogee RMA Toolkit is a Java library able to perform schedulability tests by means of simulation. The user describes system and task parameters and then lets the system run. As the system is running, the scheduler – a key component of the toolkit – assigns processor time to tasks present in the system. Tasks are assigned processor time according to their priority. The bigger the priority, the bigger the chance the task will be assigned priority. Exact rules of processor time assignment are stated in section 2.4. If there is currently no task to be executed, the system goes to idle state.

The current version of the RMA toolkit – 0.1.0 – is very limited in the way how tasks are described. They are described by a few theoretical values, see section 2.1.1. Of course, this is not sufficient for analyzing real RT-Java software. In the future, tasks will have to be described using real Java code or at least using a kind of activity description. This should allow users to analyze real software with its real execution time, not just theoretical task properties.

The output of a simulation execution is a sequence of pairs *time interval* & *task*. Every such pair says that the processor was assigned to *task* for *time interval*. The length of the interval is set by the user and it is referred to as the *preemption latency* [Dibble, 63]. It is the greatest latency that preemption can be delayed, i.e. the least time that the processor can be given to a task without interruption. When a task finishes all work in its current period, a report is made. When a task misses its deadline, a report is made too. Currently, the reports have a form of a simple message to `System.out`.

2.1 Fundamental Classes

The whole RMA toolkit consists of 15 Java classes; however, just two of them are interesting for the purpose of RMA explanation: the `Scheduler` class and the `Task` class.

2.1.1 Task

A task is a basic schedulable periodic entity. It performs some activity repeatedly, usually in a neverending loop.

It has a *period*, a *deadline*, and a *cost*. The period determines the time difference between two successive activity starts. The deadline is a time interval in which all work in the current period must be done. A task's deadline cannot be greater than its period. Implicitly, the deadline is equal to the period. The cost is an amount of work, expressed as time, that the task must accomplish in every period. The cost must not be greater than the deadline. If it is greater, the task is not schedulable.

Every task also has a *priority* which is a key element in Rate Monotonic Scheduling. The priority is either given directly by the user or it is computed during *system preparation* (see section 2.3) by the scheduler which must take into account:

- The number of available priorities.
- The least and the greatest periods of all tasks.

Therefore, if a task's priority has to be computed, it is not determined by the task itself but also by all other tasks and the environment characteristics.

After a task is created, it must be added to a scheduler using the scheduler's `addTask()` method. A task that is not added to any scheduler cannot be scheduled.

Every task can be in one of the following three states:

- *New* – A newly created task that has not been started yet is in the *New* state.
- *Wants Processor* – A task is in this state if it has not completed all work in its current period yet. The only way how to complete the work is to get processor time from the scheduler.
- *Waiting for Period Start* – A task is in this state if it has already completed all work in its current period and it is waiting for the next period start. Such process is idle and cannot be assigned processor.

At the beginning of the simulation execution, every task is started by the scheduler via its `start()` method. Inside `start()`, a task switches to the *Wants Processor* state. Subsequently, when the scheduler decides that a task will be given processor time, the task's `consumeTime()` method is called. The task decreases the amount of work to be completed in this

period. If the amount of work reaches zero before the task’s deadline, all work has been successfully done and the scheduler is informed via its `reportCompletedBeforeDeadline()` method. Then, the task switches to the *Waiting for Period Start* state. It can be switched back to the *Wants Processor* state by the scheduler when its period starts.

If a deadline is missed but there is still some time to the period end², the task switches to the *Waiting for Period Start* state as well. If a deadline is missed after next period start, the task keeps itself in the *Want Processor* state because a new amount of work must already be “consumed” by the task. In any case, a missed deadline is always reported via the scheduler’s `reportMissedDeadline()` method. The task does not report any missed deadline, the scheduler itself analyses all deadlines every time its `executeDelta()` method is called. The reason is the following: If the scheduler had not done it, but the tasks would have done it instead, some missed deadlines would not have been reported because their respective tasks would not have got any processor time and therefore a chance to check the deadline.

Since deadlines are checked during every `Scheduler.executeDelta()` invocation, a missed deadline can be reported many times until the respective task consumes all work to do.

2.1.2 Scheduler

A scheduler controls assignment of processor resources to tasks. Every scheduler knows the *number of available priority levels* M and the *least time delta* Δ_T that a task can execute for without being preempted. This Δ_T is denoted as the *preemption latency* in [Dibble, 63]. Every task to be scheduled by the scheduler must be registered first. When all N tasks are registered by the scheduler, the scheduler can compute their priorities P_i from their periods T_i , using Rate Monotonic principles. This is true in the case of automatic priority assignment only, task priorities can also be assigned manually by the user during task creation. After the priorities are known, tasks can be scheduled.

All the above activities are performed at once during *system preparation*. No other task can be registered after the system is prepared. Once preparation is completed, the scheduler can start running the system in a step-by-step

²This can never happen in version 0.1.0 of the RMA toolkit because deadlines are always at period ends.

manner. Every step lasts for Δ_T abstract time units. System preparation is discussed in detail later in section 2.3.

A step is executed using the `executeDelta()` method. The scheduler first selects a runnable task with the highest priority using the `selectNextTask()` method and gives it control for Δ_T time units. The task does its computation and therefore reduces the the remaining work to be done before the next deadline. If there is no remaining work, the task suspends itself and assumes it will be activated by the scheduler at the beginning of its next period. If a deadline is missed or all work has been done, the task reports it to the scheduler using one of the `report*()` methods.

2.2 Naming Conventions

The RMA toolkit uses the following quantities during system preparation and later during scheduling:

Number of Tasks N . Given by the user. N can grow as new tasks are added to the scheduler using the `addTask()` method.

Index over Tasks i . $i \in 0..N - 1$

Task Periods T_i . Given by the user. Every task's period is specified during the task's creation as a parameter to its constructor.

Task Deadlines D_i . Given by the user. In this version of the RMA toolkit, task deadlines are equal to task periods.

Task Costs C_i . Given by the user. Every task's cost is specified during the task's creation as a parameter to its constructor.

Task Priorities P_i . In case of manual priority assignment, task priorities are given by the user – there is a constructor parameter for the priority. In case of automatic priority assignment, task priorities are computed using the Lehoczky & Sha Constant Ratio Algorithm [Dibble, 69-70]. The actual priority assignment then happens during system preparation.

Number of Available Priorities M . Given by the user as a parameter to the scheduler's constructor.

Index over Available Priorities j . $j \in 0..M - 1$. May also be an index over computed period delimiters. Then $j \in 0..M$.

Computed Period Delimiters L_j . L_0 and L_M are known. The rest of L_j is computed from L_0 and r using this expression: $L_j = r * L_{j-1}$. The delimiters are used for automatic priority assignment. Every task's period must fit between some two adjacent delimiters L_a and L_b . The task's priority is then a . As you may have noted, there is $M + 1$ period delimiters for M priority levels. That's because $M + 1$ period delimiters, including L_0 and L_M , constitute M intervals between them.

Minimum Period Delimiter L_0 . Computed as $L_0 = \min(T_i)$.

Maximum Period Delimiter L_M . Computed as $L_M = \max(T_i)$.

Ratio of Two Adjacent Computed Periods r . Computed as $r = \sqrt[M]{L_M/L_0}$. The ratio r is constant for any two adjacent period delimiters L_j and L_{j+1} . Moreover, the same ratio r can be used to express the relation between two adjacent intervals; one interval between L_j and L_{j+1} and the other between L_{j+1} and L_{j+2} . The equality is shown later in section 2.3.2.

Schedulability Loss l . Computed from r . If $r < 2$, then $l = 1 - \frac{\ln \frac{2}{r} + 1 - \frac{1}{r}}{\ln 2}$. Otherwise $l = 1 - \frac{1}{r * \ln 2}$.

Usage of Computed Periods U_j . Computed. U_j is incremented every time a task is assigned priority j . After all tasks have their priorities assigned, U_j tells how many tasks use priority j . Also, U_j can be used to detect unused priority levels.

2.3 Priority Assignment

As already said before, there are two possibilities how a task can be assigned its priority:

- Manually by the user when the task is created.
- Automatically by the scheduler during system preparation. The priority is computed by the Sha & Lehoczky algorithm [Dibble, 69-70].

The `Task` class has two constructors that differ in number of parameters. In case of a task with manually assigned priority, the last parameter is the priority requested by the user. This parameter is missing in the other constructor version.

The highest priority is always 0. The greater the priority number is, the lower priority it represents.

2.3.1 Manual Priority Assignment

The process of manual priority assignment is straightforward. The priority is assigned in the constructor:

```
public Task(String name, int period, int cost, int priority)
```

Later, when a task is added to a scheduler using its `addTask()` method, the priority assignment type of the scheduler and that of the task are compared. If they differ, an exception is thrown out. Also, an exception is thrown out if the task's assigned priority is greater than the scheduler's greatest allowed priority number, i.e. the priority does not fit into the scheduler's allowed range of priorities.

The scheduler must be created with the `Task.ASSIGNMENT_MANUAL` value as the last argument to the constructor.

2.3.2 Automatic Priority Assignment

Automatic priority assignment is used when the priority is not explicitly determined in the constructor:

```
public Task(String name, int period, int cost)
```

During task addition to a scheduler, both priority assignment types are again compared and an exception can be potentially thrown out. The actual priority assignment is postponed until the system preparation phase.

The scheduler must be created with the `Task.ASSIGNMENT_AUTOMATIC` value as the last argument to the constructor.

When the scheduler's `prepare()` method is called, it delegates all work to `prepareAutomatically()`. This method does the following sequence of operations:

1. RM period delimiters L_j are evaluated. Given M priority levels, there are $M + 1$ period delimiters. The least priority delimiter L_0 is the minimum of all tasks' periods:

$$L_0 = \min(T_i).$$

The greatest priority delimiter L_M is the maximum of all tasks' periods:

$$L_M = \max(T_i).$$

2. The ratio of adjacent computed periods r is evaluated as

$$r = \sqrt[M]{L_M/L_0}.$$

In this point, we differ a little bit from the description in [Dibble, 70] because we believe that points 4 and 5 are not completely correct.³

An interesting phenomena can be seen here. Not only that every adjacent period delimiters L_j and L_{j+1} have the same ratio r . The same ratio r can be found between every pair of adjacent intervals later used for priority assignment. First, the delimiters L_j :

$$\begin{aligned} L_1 &= r * L_0 = r^1 * L_0 \\ L_2 &= r * L_1 = r^2 * L_0 \\ L_3 &= r * L_2 = r^3 * L_0 \\ L_4 &= r * L_3 = r^4 * L_0 \end{aligned}$$

Etc...

Then, the intervals:

$$\begin{aligned} L_1 - L_0 &= L_0 * r^1 - L_0 * r^0 = L_0 * (r - 1) * r^0 \\ L_2 - L_1 &= L_0 * r^2 - L_0 * r^1 = L_0 * (r - 1) * r^1 \\ L_3 - L_2 &= L_0 * r^3 - L_0 * r^2 = L_0 * (r - 1) * r^2 \\ L_4 - L_3 &= L_0 * r^4 - L_0 * r^3 = L_0 * (r - 1) * r^3 \end{aligned}$$

Etc...

³Specifically, we changed the statement $L_{255} = L_0 * r^{256}$ to $L_{256} = L_0 * r^{256}$ in point 4 and $r = (L_{255}/L_{0(0)})^{1/256}$ to $r = (L_{256}/L_0)^{1/256}$ in point 5.

3. The schedulability loss l is evaluated. If $r < 2$, then

$$l = 1 - \frac{\ln \frac{2}{r} + 1 - \frac{1}{r}}{\ln 2}.$$

Otherwise

$$l = 1 - \frac{1}{r * \ln 2}.$$

The schedulability loss is not used at all during actual scheduling, it is just a helper characteristics telling how much processor time can be used for useful work. The smaller the loss is, the better the processor is used. The loss is always between 0 and 1.

4. All inner period delimiters L_j between L_0 and L_M are evaluated using this recurrent expression:

$$L_j = r * L_{j-1}.$$

Theoretically, when the last L_j is computed (when $j = M$), it should equal to L_M . In practice, there is a small difference caused by implementation of real numbers.

5. Finally, all task priorities P_i are determined. To find the priority P_i of task i means to find the least j such that $T_i < L_{j+1}$. There is just one singularity: If T_i is equal to L_M then the priority is $M - 1$, i.e. the least priority.

The method responsible for finding the right priority is

```
int findRightPriority(int taskPeriod, double[] Lj)
```

6. Priority usage array U_j is computed during priority assignment in the last step. U_j is incremented at j^{th} position when priority j is assigned to a task. U_j tells how many tasks were assigned priority j .

After that, all tasks are started using their `start()` method. From now on, the system is ready for scheduling.

2.4 Execution

We recommend to test the system until the end of its critical zone is reached. Typically in a `while` loop as follows:

```
criticalZone = scheduler.getCriticalZoneLength();
while ((time = scheduler.getCurrentTime()) <= criticalZone)
    scheduler.executeDelta();
```

Obviously, this pattern can be applied only if all tasks are started at the same moment. Otherwise the Critical Zone Theorem cannot be used.

Inside `executeDelta()`, the scheduler first checks all deadlines. If a missed deadline is found in `checkAllDeadlines()` – if a task is still running after its deadline – it is reported. Then all tasks being at their period starts are woken up, using `wakeUpAllReadyTasks()`. Finally, a task to run is chosen in `selectNextTask()` and given processor time for Δ_T abstract time units by calling its `consumeTime()` method. Afterwards, the model time must be advanced by Δ_T a.t.u.

The activity inside a task's `consumeTime()` method involves decrementation of the amount of work to be done in the task's current period by Δ_T and a possible state switching, if all work has been already completed. If a task switches to the *Waiting for Period Start* state, a new wake-up event is added to the scheduler. The scheduler checks for wake-up events during every call to `executeDelta()`. If an event with the current time is found, its process is switched back to the *Wants Processor* state and the event is destroyed.

The `executeDelta()` method prints out information about the current time and the task that holds processor. This information is normally printed to `System.out` and can be redirected to a file using standard means of the used OS.

The output information may look like this:

```
Time = 48; Task A
Time = 49; Task A
Time: 49.
Task 'Task A' has successfully completed all work
    before its deadline at '100', reported at '50'.
Time = 50; Task B
Time = 51; Task C
Time = 52; Task C
Time = 53; Task B
```

Don't forget to filter out unnecessary logging information to a file using the supplied `logging.properties` file:

```
java -Djava.util.logging.config.file=logging.properties
    Dibble90Automatic
```

2.4.1 Complexity

The real time spent for simulation grows *linearly* in the number of simulation steps necessary to reach the end of the critical zone. The longer the critical zone is, the more steps necessary. On the other hand, the number of steps can be reduced by increasing the preemption latency Δ_T .

In every simulation steps, the time necessary to select a task grows *worse than linearly* in the number of task. The algorithm in `selectNextTask()` goes twice through the task list using an iterator – this is of linear complexity. In addition, the list is ordered using `Collections.sort()`, which is of complexity $\mathcal{O}(n * \log n)$ where n is the number of tasks.

Therefore, the final complexity of the whole simulation until the end of the critical zone is of complexity $\mathcal{O}(s * (n + n * \log n))$ where s is the number of necessary steps and n is the number of tasks.

3 Future Extensions

The toolkit presented in the previous section is our first attempt to provide a solution for schedulability analysis of real-time programs. We chose a theoretical approach for the beginning and we plan to extend the toolkit in the future towards practical usage, i.e. towards analysis of real Java code.

3.1 Respecting Variations of Task Execution Time

The simulation approach to the RMA analysis presented within this report can be straightforwardly extended for randomly generated task execution times instead of deterministic constants (maximum duration of execution) that are used here. An open question remains how long to let the model execution run in order to reach a chosen reliability level of the final assertion schedulable or not schedulable. The Critical Zone Theorem [Dibble, 90] is not valid anymore because the system load in the critical zone need not necessarily be the worst one.

3.2 Handling Task Synchronization

RMA schedulability analysis, in the simple form presented above, does not assume any dependence of tasks execution. Clearly, it is far from reality, because tasks need to interact in order to reach the goal of multithreaded computation. General model of Java multithreaded computation uses indirect interaction of threads, i.e. threads act as clients that call services – synchronized methods – of dedicated passive objects – monitors. A thread can be blocked when calling a monitor service, because the service – implemented as a monitor synchronized method – need not be available depending on the monitor state, i.e. the data encapsulated within the monitor. A thread can be blocked when waiting for an external event.

The presented reasons generally prevent us from taking a thread execution as a continuous activity and, as a consequence, to estimate precisely the overall time of task execution (one parameter of the RMA method).

One possible way how to overcome this limitation is to use the model-based method of Java reactive program evaluation, as presented in [SimCheck]. The core of the method is to run real Java program code together with a model of the program environment. Using the J-Sim simulation library and the model-time concept, the execution can be deterministically serialized

with preservation of its time relations. Then the thread schedulability analysis can be done in a similar way as presented here in chapter 2, but with data dependencies and task interactions taken into account as well. Moreover other properties of the program behavior can be investigated using model-based test cases.

3.3 Aperiodic Tasks

It should not be a hard task to add support for aperiodic tasks. An aperiodic task does not have a period because it executes just once and disappears afterwards. Aperiodic tasks are usually handled by an *aperiodic server*, as described in [RMA, 7]. Such a server has an execution budget and a replenishment period. As long as there is some time in the execution budget, aperiodic tasks can execute. As soon as it is spent, aperiodic tasks cannot run or they can run only at a low priority. The execution budget is gained back after the replenishment period elapses. The details how aperiodic tasks are run if the budget is spent and how it is gained back (how the replenishment period is treated) differ according to the exact type of the used aperiodic server.

Of course, support for aperiodic tasks would require the ability to add tasks to the system even if it is running already, not just before system preparation. This feature would be useful for periodic tasks too, or at least postponed task starts. The automatic priority assignment would not always work in case of tasks added to the system after preparation. The new task's period being shorter than L_0 or longer than L_M would require the scheduler to re-evaluate all characteristics and (possibly) to re-assign task priorities. This is not a desired behavior, RMA uses static priorities assigned before the program starts.

3.4 Execution Time Detection

The present version of model-based Java reactive program evaluation uses an estimated value of duration of locally executed thread code. Thread execution dynamics is then taken as a sequence of locally executed parts of code with known time of execution interleaved with time intervals when the thread can wait for an event, e.g. for renewing monitor ability to provide a requested service. The duration of these intervals is apriori unknown and can be computed on-line when the checked program is executed within the

J-Sim based model environment.

The model-based method of reactive Java program checking can be improved when using measured time durations instead of estimated time durations of locally executed parts of a thread's code. The duration can be measured on-line, i.e. during the model-based (i.e. serialized) program execution. The measured (real-time) values need to be recomputed, respecting relative speeds of the processor executing the model and the target processor intended to be used, and used as model-time durations of threads' locally executed parts.

3.5 GUI

Model based execution of multithreaded Java program either in the simple form presented within this report or in the more sophisticated (intended) version can be visualized and interactively managed in order to provide the user with a more convenient interface.

4 Conclusion

The report has presented a simulation-based method of RMA (Rate Monotonic Analysis). A simulation program was implemented in Java using the RMA toolkit as its background. The presented version of the simulation program uses a simple (console based) user interface. The program copes with a set of tasks, each of them is described by three parameters: priority, maximum time of execution, and planning period taken as task execution deadline as well. The parameters are given in a form of a user-written Java class. The program decides if the given set of tasks is or is not schedulable, i.e. if all the tasks meet their requested deadlines or not. Possible improvements and extensions of the simulation approach to the RMA were analyzed and they are intended to be subjects of our future work.

The toolkit is available on-line at <http://home.zcu.cz/~jkacer/en/software/rma-toolkit/>. Access to this site is protected by a username and a password. The installation archive includes full source code, compiled classes, API documentation, and two examples – the example from [Dibble, 90] with automatic and manual priority assignment. The result of the manual priority version are exactly the same as in the book. In case of the automatic assignment, the results differ because tasks B and C are both assigned the lowest priority and the medium priority remains unused.

References

- [RMA] *Lui Sha, Mark H. Klein, John B. Goodenough: **Rate Monotonic Analysis for Real-Time Systems***. Technical Report CMU/SEI-91-TR-6, ESD-91-TR-6. Carnegie Mellon University, Software Engineering Institute, Pittsburgh, U.S.A., 1991.
- [Dibble] *Peter C. Dibble: **Real-Time Java Platform Programming***. Sun Microsystems Press, Palo Alto, CA, U.S.A., 2002. ISBN 0-13-028261.
- [SimCheck] *Jaroslav Kačer: **Simulation-Based Checking of Java Concurrent Programs***. Ph.D. Thesis, University of West Bohemia in Pilsen, FAV-KIV, Pilsen, Czech Republic, 2005.