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

Simulation-Based Schedulability Analysis of RMA Programs

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 Schedulability Analysis of RMA Programs

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.

Contents

List of Figures

1 Introduction

Aaa

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 assignes 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 states 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:

- 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 completes all work in its current period and it 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 end[1], 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 asigned 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

[1]This 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 that 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 that 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 $l = 1 - \frac{ln\frac{2}{r}+1-\frac{1}{r}}{ln2}$. Otherwise $l = 1 - \frac{1}{r*ln2}$.

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 assignes, U_j tells how many tasks use priority j. Also, U_j can be used to detect unused priority levels.

2.3 Priority Assignment

Aaa

2.3.1 Manual Priority Assignment

Aaa

2.3.2 Automatic Priority Assignment

Aaa

2.4 Execution

Aaa

3 Future Extensions

The toolkit presented in the previous section is our first attempt to provide a solution for schedulability analysis of RMA 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.

!!!!!!!!!! To se mi nezda !!!!!! Jarda. 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.

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 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 J-Sim simulation library and the model-time concept, the execution can be deterministically serialized with preservation of its time relations. Then the threads schedulability analysis can be done 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 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.

3.4 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 paper presents a method of model based development and (partial) verification of concurrent

References

[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.