Design Pattern for the Adaptive Scheduling of Real-Time Tasks with Multiple Versions in RTSJ

Rodrigo Gonçalves, Rômulo Silva de Oliveira, Carlos Montez
LCMI – Depto. de Automação e Sistemas – Univ. Fed. de Santa Catarina
(rpg, romulo, montez)@das.ufsc.br

Abstract

This paper presents a design pattern of an adaptive scheduling based on the management of the tasks execution time, achieved through multiple versions of the tasks, applied to the Real-Time Specification for Java. A structure of classes is used to facilitate the development of tasks, while allowing the independence of the application code from the code responsible for the adaptive control. The design pattern is described through UML diagrams and an example implementation is presented.

1. Introduction

The Real-Time Specification for Java (RTSJ) is an extension of the standard Java platform that is able to attend the restrictions imposed by real-time systems, such as predictability and determinism [1, 2, 3]. The RTSJ adds to the Java standard the following characteristics: it adds real-time threads, it allows the execution of code without the interference of the garbage collector, it also allows the control of the localization of objects and the access to memory physical addresses; it implements a manager of asynchronous events and a mechanism for asynchronous transference of control between threads [1, 2, 3].

An area where the RTSJ will be very important in the next years is the network control. In this type of application, the control of industrial processes is made through a network. Sensors, controllers and actuators are spread by diverse processors in a distributed environment. The control of the industrial process presents real-time requirements and it is affected by the delays in the communication network and the involved processors.

A way to deal with the inherent difficulties to the fulfillment of real-time requirements in distributed environments is the adaptive scheduling. In dynamic environments that are managed by adaptive scheduling, the conditions in which the tasks are executed are defined during the execution of the application itself. One of the forms described in literature to provide such adaptability is to allow tasks to have many versions. In the case of network control, there can be two versions of the controller, a fast one but of inferior quality, and another one with maximum quality but increased execution time. The execution time of the task can be dynamically modified in run-time.

The objective of this work is to elaborate a design pattern defining a class structure capable to encapsulate the code associated with the multiple versions of a task (code that implements different versions of an algorithm). It will facilitate the implementation of new tasks, at the same time it makes the application algorithm independent of the adaptation mechanism. Thus, a programmer can specialize itself only in the development of the application, not having knowledge of how the adaptive control was implemented internally. Another programmer will be responsible for the control classes and the definition of the scheduling algorithm to be used. The main goal is to facilitate the development of real-time applications that apply adaptive scheduling, making possible the reuse of code and allowing the separation of the scheduling algorithm from the application tasks. The use of design patterns represents reduction of costs and greater agility in the implementation.

The next sections present the design pattern considered for the adaptive scheduling based on the management of the tasks execution time. UML diagrams show the structure of the solution and its relationships, as well as the events waited during the execution of the system. Experiences had been carried through to evaluate the behavior of this solution and the result is shown.

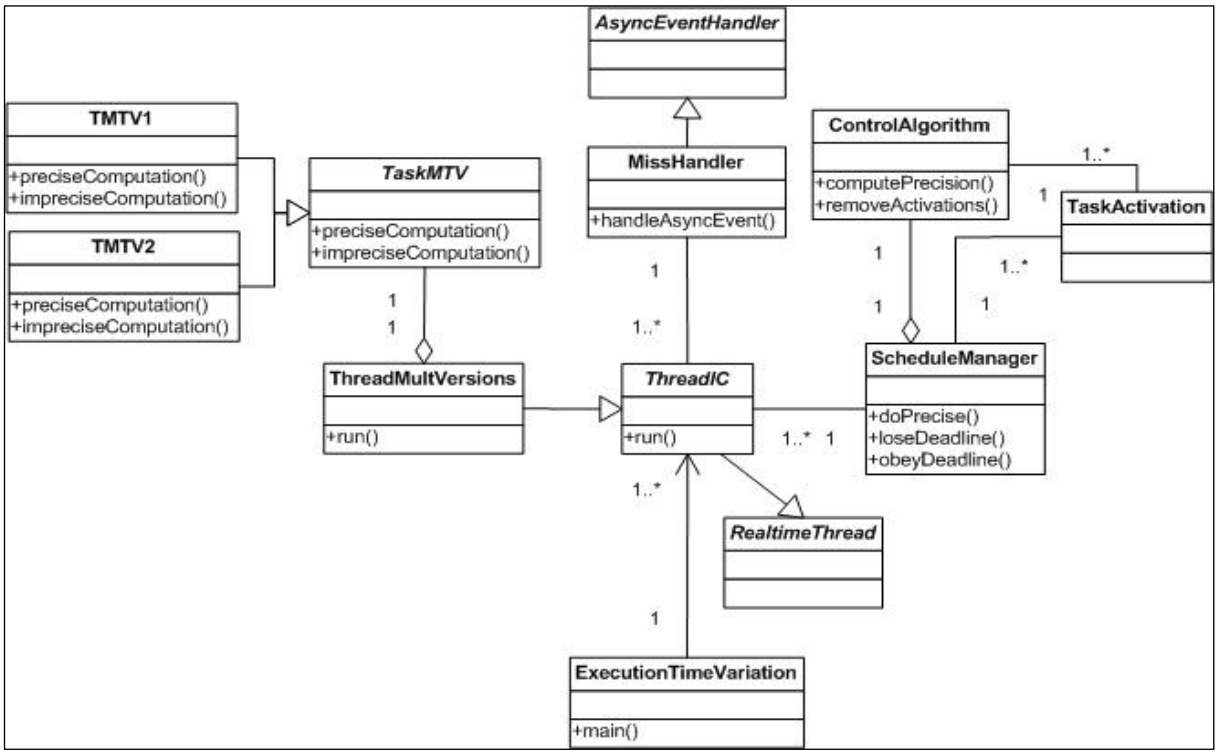

Figure 1. UML class diagram for the proposed solution.

2. Adaptation of the Execution Time in RTSJ Based on Multiple Versions

The solution proposed for the management of the execution time is shown by the UML class diagram of figure 1. The method *main()* is implemented in the class *ExecutionTimeVariation*, where it initiates the execution of the application by creating *threads* with multiple versions. The initialization happens according the sequence of events shown in figure 2. Following, figure 3 shows a small part of code of this class with the creation of imprecise *threads*.

The solution proposed utilizes periodic real-time threads. However, the proposal presented in this paper, and described by figure 1, does not hinder the utilization of others typed of tasks, such as aperiodic or sporadic. We just opted for periodic tasks since them facilitate the observation and analysis of the adaptation of the tasks.

Figure 2 shows the events of the instantiation of an imprecise real-time task. The mechanism of multiple versions is implemented by the *ThreadMultVersions* class, that is a specialization of the *ThreadIC* class (it initializes the parameters of threads with multiple versions), and its objects are instantiated by the *ExecutionTimeVariation*. The *ThreadMultVersions* class supplies the necessary structure for the execution of the real-time task, by consulting the scheduling on which version of the task it will be carried through, the precise or the imprecise version.

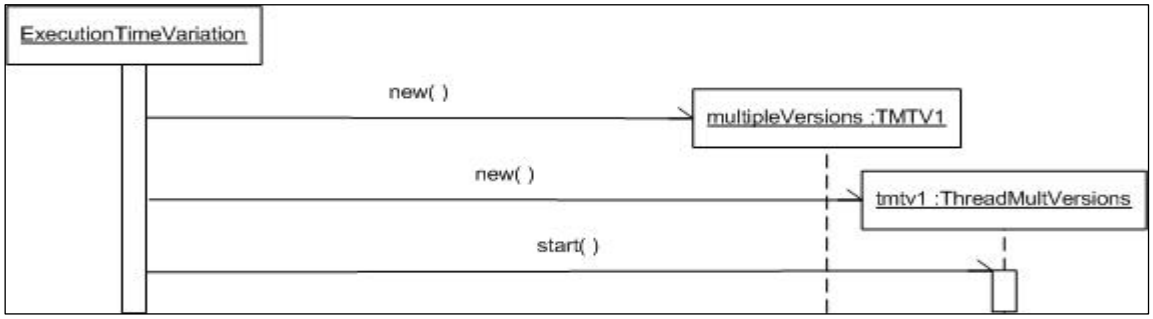

Figure 2. UML sequence diagram of the creation of a thread with multiple versions.

```
//Class ExecutionTimeVariation: creating imprecise real time threads
//(in this case, periodic tasks had been used). Parameters: begin,
//period, cost, deadline, priority, task ID and task object
//(a subclass of TaskMTV)
ThreadMultVersions tmtv1 = new ThreadMultVersions(1000, 2000, 2000,
   2000, 10, 1, new TMTV1());
ThreadMultVersions tmtv2 = new ThreadMultVersions(2000, 4000, 4000,
   4000, 20, 2, new TMTV2());
tmtv1.start();
tmtv2.start();
```

Figure 3. Part of the code of class *ExecutionTimeVariation*.

Figure 4 shows the constructor of the class *ThreadIC*, which inherits properties from the class *RealtimeThread*. Its utility is to simplify the programming of the imprecise tasks, since they share the same code implemented in this class, whose function is to construct a real-time thread with the supplied parameters. Figure 5 shows the constructor of the class *ThreadMultVersions* that calls the constructor of the superclass and associates the imprecise task to be executed by the thread.

```
//Constructor of the class ThreadIC, parameters: task ID, begin, type
//of the task, period (in this case, it's a periodic thread),
//computational cost, deadline and priority.
public ThreadIC(int id, long begin, String tTask, long period,
   long cost, long deadline, int priority) {
   (...)
   //ReleaseParameter with parameters (periodic thread, period, ...)
   ReleaseParameters release = new PeriodicParameters(
       new RelativeTime(begin, 0),//start
       new RelativeTime(period, 0),//period
       new RelativeTime(cost, 0),//computational cost
       new RelativeTime(deadline, 0),//deadline equals to period
       null,//overrun handler
       new MissHandler(this, tTaask)//miss handler
   );
   // SchedulingParameter with it's priority
   SchedulingParameters sched = new
       PriorityParameters(PriorityScheduler.MIN_PRIORITY + prioridade);
   //Defining parameters of the thread
   setSchedulingParameters(sched);
   setReleaseParameters(release);
}
```

Figure 4. Code for the constructor of class *ThreadIC*.

```
//Constructor of the class ThreadMultVersions, parameters: task ID,
//begin, period, computational cost, deadline, priority and a
//reference to the abstract class TaskMTV, superclass of the
//imprecise task to be executed - TMTV1.
public ThreadMultVersions(int id, long begin, long period, long
   cost, long deadline, int priority, TaskMTV mtv) {
   //Calling the superclass constructor
   super(id, begin, "Thread Multiple Versions", period, cost,
      deadline, priority);
   //Associating multiple version task to be executed
   multipleVersions = mtv;
}
```

Figure 5. Part of the constructor for class *ThreadMultVersions*.

The constructor of *ThreadIC* (figure 4) declares the release parameter (*ReleaseParameters*) of the thread as *PeriodicParameters*, which defines the periodic characteristic of the task. The one before last parameter used in the declaration of the object *release* is called the overrun handler (*OverrunHandler*). It is executed when the JVM detects that one thread spent more time in the computation than the value defined as its *cost*. However, this function is not implemented by the virtual machine used in the experiences. The last parameter is the *MissHandler*, it answers to an event generated by the JVM when one thread does not finish its execution until the limit defined as its deadline. One of the parameters of this handler is a reference to the thread that caused the event.

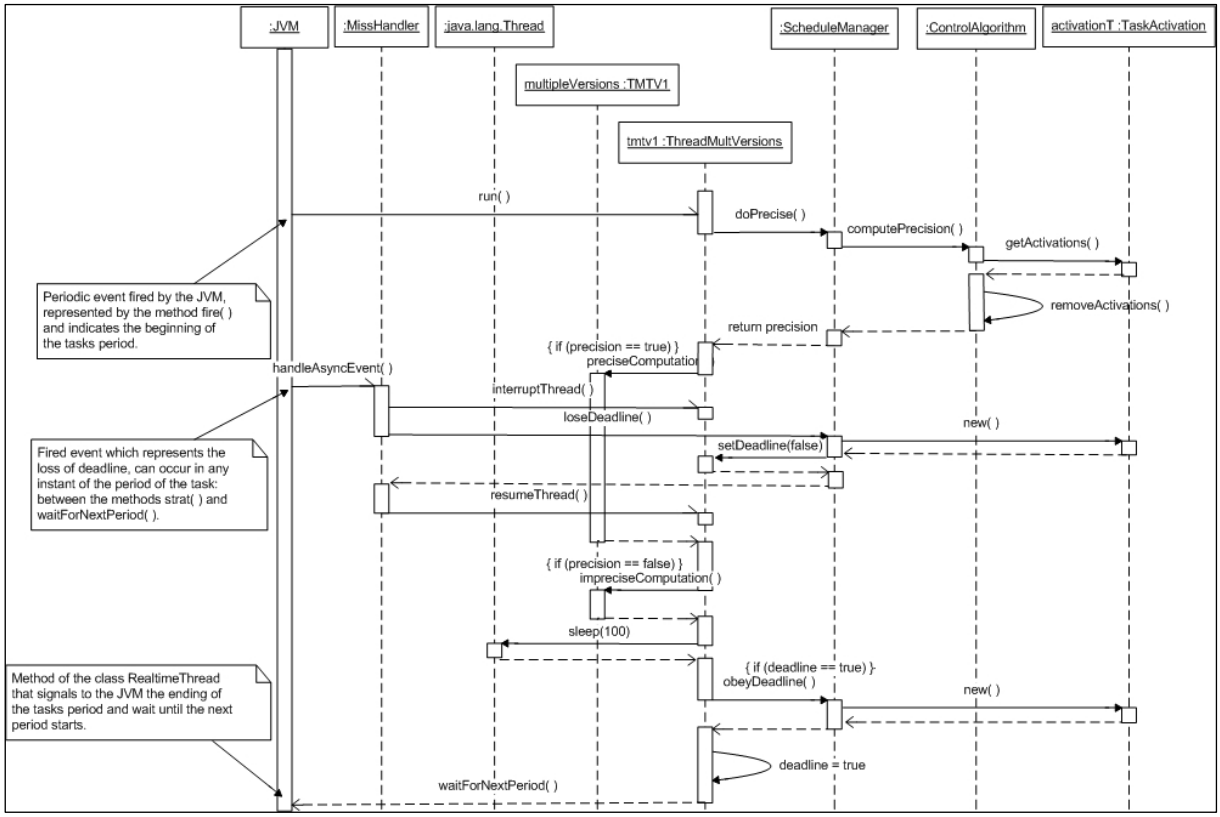

Figure 6. UML sequence diagram for *ThreadMultVersions*.

```
//The run() method implements the ThreadMultVersions execution:
//compute the number of activations, consult the adaptive scheduler
//of which version will be executed and warns the scheduler if the
//task deadline was fulfilled or not.
public void run(){
   do {
       //Task activation control
       activation++;
       //ScheduleManager defines which version will be executed
      if(ScheduleManager.doPrecise())
        multipleVersions.preciseComputation();
      else
        multipleVersions.impreciseComputation();

       //Warn the fulfillment of the deadline to the ScheduleManager
       if(dead)
         ScheduleManager.obeyDeadline(typeTask, taskID, activation);
       //Variable dead has to be defined with the value true, which
       //refers to the next task activation.
       dead = true;
   }while(waitForNextPeriod() && activation < 30);
}
```

Figure 7. Code for method *run( ) of ThreadMultVersions*.

The constructor of the class *ThreadMultVersions* (figure 5) receives as last parameter a reference to the task to be executed, an object of class *TMTV1*, and associates this task to the thread just created. This class (*TMTV1*) is derived from the abstract class *TaskMTV*.

Figure 6 represents an UML sequence diagram for a period of *ThreadMultVersions*. It should be noticed that the algorithm initiates consulting to the *ScheduleManager* on which precision level should be carried through by the task, that then executes the precise or imprecise version of *multipleVersions* (class *TMTV1*). At the end, the thread is suspended for a small interval and then it verifies if the deadline was missed (through variable *dead*). If the deadline was not missed the *ScheduleManager* is informed of the meeting of the deadline and information about this activation is stored in a object *TaskActivation*. During the execution of the thread, the JVM can detect the deadline miss, in this case it calls the class *MissHandler*, which is responsible for managing these events. Class *MissHandler* interrupts the execution of the task and informs the *ScheduleManager* of the deadline miss. The method *waitForNextPeriod()* is the last event, which informs to the *JVM* about the ending of the period and prepares itself for the next activation. Figure 7 shows the code of the class *ThreadMultVersions* periodically executed.

The imprecise tasks implemented by objects *multipleVersions* are determined at the threads initialization in the class *ExecutionTimeVariation* as objects of class *TMTV1*. Class *TaskMTV* is abstract and serves as an interface for the imprecise tasks, as showed by figure 1.

Class *ScheduleManager* plays a parallel role to the one of the class Scheduler. The objective of the *ScheduleManager* is to concentrate the process of adaptive scheduling, since that tasks call this class to determine which precision level should be used in their execution.

In order to guarantee mutual exclusion during the access to methods of the class *ScheduleManager*, they are implemented with the directive *synchronized*. It should be noted that some methods receive a *RealtimeThread* reference for the current thread as a parameter, since they need to have access to the imprecise threads, as in *((ThreadIC) th).setDead(false)*, whose function is to inform the task about a deadline miss. Figure 8 shows the code of *ScheduleManager*.

```
//Methods of the class ScheduleManager
public class ScheduleManager {
   private static RealtimeThread rtt;
   private static ThreadFire tf = new ThreadFire();
   //Task activations list
   private static LinkedList listaJanelaAT = new LinkedList();

   public synchronized static boolean doPrecise() {
      //ControlAlgorithm determines the next task activation
      return ControlAlgorithm.computePrecision(listaJanelaAT);
   }
   public synchronized static void loseDeadline(RealtimeThread th,
       String tTask, int id, int activ) {
       //Create the activation registry, the deadline was lost
      TaskActivation activationT = new TaskActivation (taskID, activ,
        false);
      listaJanelaAT.add(activationT);
      //Warning the task that it's deadline was lost
       ((ThreadIC) th).setDead(false);
   }
   public synchronized static void obeyDeadline(String tTask,
       int taskID, int activ) {
       // Create the activation registry, the deadline was fulfilled
      TaskActivation activationT = new TaskActivation (taskID, activ,
        true);
      listaJanelaAT.add(activationT);
   }
}
```

Figure 8. Part of the code for class *ScheduleManager*.

3. Example of an adaptive scheduling algorithm

Class *ControlAlgorithm* is responsible for deciding if the adaptation will be applied to the task. In the implementation of class *ControlAlgorithm* the adaptation algorithm used is very simple. Only the characteristics of the activations (object *TaskActivation*) were considered by the scheduling algorithm. More complex algorithms probably would use more information. For example, descriptors for the threads managed by the system with all the data of each task.

Objects of the class *MissHandler* are associated to imprecise threads in the constructor of *ThreadIC*. The purpose of these objects is to manage the deadline misses and to inform to the *ScheduleManager* so that the adaptation can be applied to the task. The constructor of *MissHandler* assigns the maximum priority to the *MissHandler* thread. The method *handleAsyncEvent()* is called by the *JVM* when the deadline miss is detected and its function is to inform the *ScheduleManager* of the deadline miss (*loseDeadline()*). Then it reschedules the task again, with the method *schedulePeriodic()*. Figure 9 shows the implementation of class *MissHandler*.

```
//Class MissHandler, responsible to manage deadlines misses
public class MissHandler extends AsyncEventHandler {
   //Thread to associated to manages deadlines misses
   RealtimeThread me;

   public MissHandler(RealtimeThread th, String tTask) {
      //Defining the highest priority to the MissHandler
       super(new PriorityParameters(PriorityScheduler.MAX_PRIORITY),
          null, null, null, null, null);
      //Recovering data of the task who missed the deadline
       me = th;
       typeTask = tTask;
   }
   //Method that manages the deadline missed
   public void handleAsyncEvent() {
      //Warns to ScheduleManager of the deadline missed
      ScheduleManager.loseDeadline(me, typeTask,
          ((ThreadIC) me).getID(), ((ThreadIC) me).getActivation());

       //Reschedule the thread who missed the deadline
       me.schedulePeriodic();
   }
};
```

Figure 9. Part of the code of class *MissHandler*.

4. Experiences

As said previously, the implementation of the tasks with multiple versions was developed for periodic threads, since this facilitates the analysis of the adaptation of the imprecise tasks. The task parameters are defined during the initialization of the threads in class *ExecutionTimeVariation*. The most relevant parameters used in the experiences are the same for all the tasks: start at 1000 ms, period of 2000 ms, computation time of 2000 ms, priority 10 and the specific task with multiple versions.

The computation done by the application tasks is only a simulation of some mathematical algorithm with the objective of consuming processor time until a deadline is either missed and, then, the task is submitted to a variation of its execution time. The computational cost of the task is increased at each period until, as consequence of a deadline miss, the adaptation calls the imprecise version of the task and its computational cost is reduced to avoid future deadline misses.

The adaptation algorithm in class *ControlAlgorithm* stores the states (deadline met or missed) of the last activations, defined by the variable *activationWindow*. The adaptation algorithm consults these activations to determine the precision level of the task to be executed next. If some activation within *activationWindow* lost its deadline, then the imprecise version of the task will be executed. The graphic in figure 10 shows the adaptation of a task implemented with multiple versions.

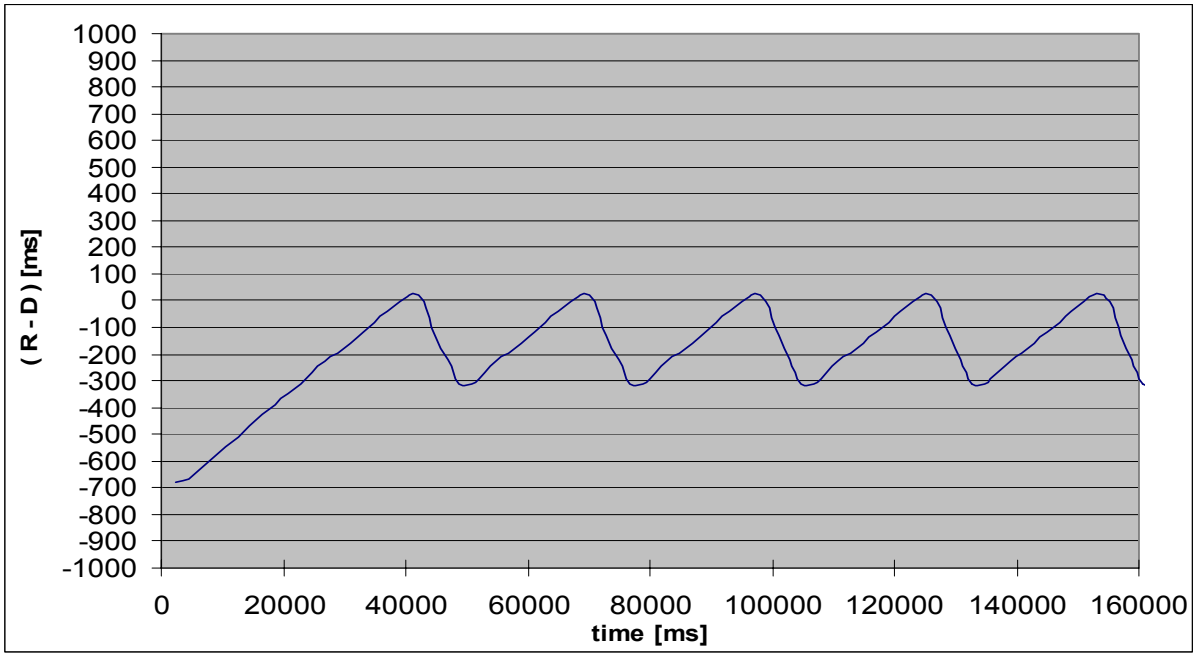

Figure 10. Example of an imprecise task with multiple versions.

The graphic shows the difference between the conclusion time and deadline of the task at the vertical axis, as a function of the time, which is at the horizontal axle. While this difference has a negative value it indicates that the task is meeting its deadline. But at the instant that this value becomes positive, it is necessary to adapt the execution time of the task so as it will meet the temporal requirement again.

A problem observed was that, as the application task does a mathematical calculation, the processor utilization by the task is high and constant. During some tests it was observed that by the ending of a period, in which the deadline was missed, the event created to signal this problem does not have a chance to be handled, because the next activation of the task starts its execution. This is a problem that the Java Virtual Machine used presents when facing asynchronous events (*AsyncEventHandler*) and priority inversion (the higher priority task does not interrupt the execution of the lower priority task). It was confirmed by *Corsaro* et. al. [9]. In order to give a chance for this event to be treated it was included a *sleep()* call with the objective to suspend the current thread.

5. Conclusions

The objective of the design pattern proposed in this paper was to facilitate the development of real-time tasks for adaptive scheduling based on the variation of the execution time.

In some situations it is not possible to use periodic real-time threads, implemented as *RealtimeThread* objects with the *PeriodicParameter* parameter, in the Java Virtual Machine of the Reference Implementation (*JVM-RI* [5]). Many of its methods are not implemented correctly (for example, the method responsible for the change of a task period). However, whenever it is possible to use this type of thread, the development of periodic real-time tasks becomes much more simple.

The design pattern described in this paper was defined and planned at the application level. The adaptation only actuates on the tasks created by the application itself. However, some classes of this proposal execute functions very related to the services of the operational system or of the *JVM*. For example, classes *ScheduleManager* and *ControlAlgorithm*, whose functions are similar to the *JVM* scheduler. In future works new schedulers with adaptation based on imprecise tasks can be developed and be incorporated to the Java Virtual Machine.

Finally, it can be noticed that the use of the proposed class structure, as described by the diagram in figure 1, allows the development of tasks based on the multiple version technique and the objective of this work was accomplished. Analyzing the results it is observed that the tasks, executed periodically, are adapted always that the *JVM* detects a deadline miss, deciding for the execution of the imprecise version of the tasks as a way to fulfill its temporal requirements.

6. References

[1] G.Bollella, B.Brosgol, P.Dibble, *et al*. *The Real-Time Specification for Java.* Addison-Wesley, 2001.

[2] P.C.Dibble. *Real-Time Java Platform Programming*, Prentice-Hall, 2002.

[3] A. Wellings, *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[4] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*. 1st Ed.. Addison-Wesley, 1998.

[5] *TimeSys Corporation* (*Embedded Linux & Development Products*): www.timesys.com. January, 2005.

[6] L. Chenyang, J.A. Stankovich, T.F. Abdelzaher et. al. *Performance Specifications and Metrics for Adaptive Real-Time Systems - RTSS*. Florida, 2000.

[7] J.W.S. Liu, K.-J. Lin, W.-K. Shih, et. al. *Algorithms for Scheduling Imprecise Computations.* In: IEEE Computer Society Press, CA, 1991.

[8] J.W.S. Liu, K.-J. Lin, W.-K. Shih, et. al. *Imprecise Computations.* In: Proceedings of the IEEE, January, 1994.

[9] A. Corsaro, S. Douglas, *The Design and Performance of Real-time Java Middleware*. In: Transactions on Parallel and Distributed Systems, November, 2003.