Design and implementation of a Maple-package for the predictability of real-time systems

Mario Korte, Karsten Albers, Frank Slomka
Department of Computer Science, University of Oldenburg
Ammerländer Heerstr. 114-118
D-26129 Oldenburg, Germany

$$\left.\begin{array}{r}\textbf{mario.korte}\\ \textbf{karsten.albers}\\ \textbf{frank.slomka}\end{array}\right\}\textbf{@informatik.uni-oldenburg.de}$$

March 17, 2006

Abstract

Many critical systems are controlled by computers, for example airplanes, nuclear power plants, airbag systems in cars etc. The most important factor of the used systems is of course that they work error free. But for the subset of hardware and software systems named real-time systems it is as well most important, that the system works absolutely predictable in both, time and function. Working predictable in time means that all software tasks meet their previously defined deadlines. The implemented Maple package supplies test methods and functions for scheduling algorithms like EDF (Earliest Deadline First), RMS (Rate Monotonic Scheduling) and DMS (Deadline Monotonic Scheduling), that have been proved to predict the compliance with previously defined deadlines.

1 Introduction

Today computers can be found in everyones car, in many machines, spacecrafts and planes. The telephone system is build with computers and each television set uses microprocessors. However, the design of software for such embedded computer devices is different from writing software for general purpose computers on our desktop. An important design challenge of such embedded systems is the correct timing behavior of the software. Since the early days of real-time computer design a mathematical theory of embedded real-time systems was developed. This theory allows the predictable design of real-time systems. The given parameters in the theory are the worst- and best case execution time of a software task, the timing behavior of events triggering the tasks, and the given deadline in which the computation must finish to fulfill the needs of the systems application. Additionally it considers preemptive or non preemptive tasks. Important design parameters of such systems, the questions the designer will ask to his construction, are: do all software tasks meet their deadlines, which response time can be calculated for a task and what is the processor utilization. The goal for the analysis is to find a feasible scheduling of real-time software tasks on one ore more given microprocessors.

The main contribution in this field was a paper written by Liu and Layland in 1973 [1]. In this paper analysis techniques for real-time systems using priority based scheduling and earliest deadline first are presented. It was shown that both strategies are optimal and how the priority of a task can be calculated by the designer of such a system to get an optimal scheduling. However, the results are limited in some cases. These limitations are removed by later work, for priority based systems by Burns and Wellings [2], for earliest deadline first scheduling by Baruah [3] and independently by Gresser [4].

In this paper we introduce a Maple library for the analysis of real-time systems. The main insistence of this package was to use it during lectures and to give the students a tool to play with and test their own calculations on various task sets. Real-time analysis is a mathematical theory and expressing its results with a tool like Maple gives students and researchers a new access to the theory. The latest results in this field, as given by Baruah [3], Gresser [4], Chackraborty and Thiele [5], Albers and Slomka [6], are using mathematical functions which can be interpreted graphically. While Maple is a powerful tool to plot functions it becomes a new tool to explore and interpret results from the field real-time analysis. Using the real-time analysis package described in this paper it is easy to develop new analysis algorithms and to find new results in this growing field. The package is available for free to students, lecturers and researchers.

2 Real-time analysis package

In this section a short introduction to real-time analysis is given and the belonging functions of the package are described. While the main contribution of the paper is the description of the Maple package we only describe some fundamentals of the theory. A good text book in the field of the design and theory of embedded real-time systems was written by Buttazzo [11].

2.1 Definitions of tasks and task sets

In Maple the definition of data types takes place in a procedure whose input is an object to be tested and its output is a boolean value ("true" or "false"). Within the procedure the evaluation takes place, for which integrated Maple type definitions can be used as well as other formulas and equations.

A task (Maple type definition: *task*) is defined as an ordered set of mathematical expressions: the period p, the ready time r, the deadline d and the worst-case execution time c (figure 1). Since there are periodic and aperiodic

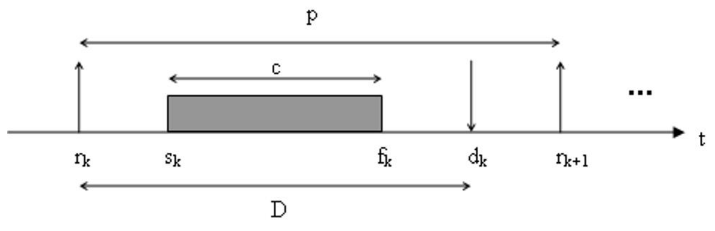

Figure 1: Illustration of a task

tasks, the period is optional. Thus a task can be defined by an ordered sequence (Maple type: *list*) with either 3 or 4 elements. These elements must be positive numbers or 0 (Maple type: *nonnegative*). A missing period is defined as the value $+\infty$. So aperiodic tasks can also be entered as periodic tasks with an infinite period. As a consequence the value of a period can be a positive number but also $+\infty$ (Maple type: *posinfinity*). If all these properties are fulfilled an object is accepted as a task.

A set of tasks (Maple type definition: *taskset*) is defined as a non-empty set of tasks. Therefore the only two properties that have to be checked are: the input object has to be of Maple type list and all elements have to be of the previously defined Maple type task.

To use these two types in the package they are defined and activated in the special module function "ModuleLoad", which is executed at loading of the module.

The first group of functions *period (p), ready (r), deadline (d)* and *wcet (c)* is used to compute and/or return the elementary task components, e.g. absolute and relative deadline, ready times of the tasks and instances as well as the period and the worst-case execution time. These functions can also be used later in Maple outside of the package to simplify using the task characteristics in own formulas and equations.

2.2 Static priority based scheduling with RMS and DMS

Liu and Layland [1] found that on one single processor preemptive software tasks having a deadline equal to the period of their invocation can be optimal scheduled, if the priorities of the tasks are selected in the following way: The task with the highest rate of triggering events, or the shortest triggering period, gets the highest priority. The tasks with the slowest rate get the lowest priority. Such a scheduling scheme is called rate monotone scheduling (RMS). This scheme can also be used if a deadline shorter than the period is given for each task. In this case the lowest priority is assigned to the task with the shortest deadline. This algorithm is called deadline monotone scheduling (DMS). For RMS the schedule is always feasible, if the utilization of the processor (package function: U) is equal or smaller then a given bound (package function: *Umax*), which depends on the number of tasks n:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq U_{max}^{RMS}(n) = n(2^{1/n} - 1)$$

In this formula the value c describes the worst-case execution time of the task and p the period, as described in the definition of tasks. For the DMS scheduling test using the processor utilization the period must be lowered to the value of the deadline, which is a very pessimistic test:

$$U = \sum_{i=1}^{n} \frac{c_i}{d_i} \leq U_{max}^{RMS}(n) = n(2^{1/n} - 1)$$

The scheduling test using the processor utilization analysis is implemented in the functions *RMStest* and *DMStest*. Unfortunately these formulas are only necessary. This means that schedulable task sets exist, which are having a processor utilization greater than this bound. To overcome the problem, Joseph and Panday [8] published a more general approach to the problem: The response time analysis (RTA). Response time analysis calculates the worst case response time of each task. The worst case response time of a task (package functions: *RMSresponsetime* and *DMSresponsetime*) depends on its priority, its worst case execution time, the number of interrupts given by higher priority tasks, and the worst case execution time of this high priority tasks. For this equation the task set must be ordered by priorities:

$$R_i^{k+1} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{p_j} \right\rceil c_j,\ R_i^0 = c_i$$

This formula must be solved iteratively, because the interrupt of a high priority task extents the time used to calculate the number of interrupts. The calculation ends if the formula convergates or if the deadline of the considered task is reached. The scheduling tests using the RTA are implemented in the functions *RMSresponsetimeTest* and *DMSresponsetimeTest*.

Additionally for RMS another test procedure is offered using the cumulated processor demand as described by Lehocky, Sha and Ding [9]. For this schedulability test the cumulative processor demand W (package function: W) for the tasks $\tau_1, \ldots, \tau_i$ in the interval $[0, t]$ is defined as:

$$W_i(t) = \sum_{j=1}^{i} c_j \cdot \left\lceil \frac{t}{p_j} \right\rceil$$

as well as:

$$L_i(t) = W_i(t)/t\ ,$$
$$L_i = min_{0<t\leq p_i} L_i(t) \text{ and}$$
$$L = max_{\forall i} L_i$$

All these functions are computed in the overloaded package function L.

The task set is feasible if: $L \leq 1$. This test is implemented in function *RMScumulativeDemandTest*.

For the formulas $W_i(t)$ and $L_i(t)$ the functions *plotW* and *plotL* are implemented, which display the graph of those functions.

2.3 Dynamic priority based scheduling with EDF

The second policy to schedule tasks in real-time systems is called earliest deadline first (EDF)[1]. In EDF the task with the shortest remaining deadline is scheduled. A task set is feasible if the processor utilization is smaller or equal 100%:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$$

This test is used in function *EDFtest*. However, this result holds only, if the deadline of each task is equal to the tasks period. This condition is not realistic in todays real-time systems. To calculate the feasibility of task sets with deadlines shorter than the task periods an approach given by Baruah [3] can be used. This approach is fundamental to real-time analysis and is called the demand bound function (package functions: *singleDBf* and *DBf*):

$$D_b(T,I) = \sum_{\substack{i=1}}^{\overset{I \geq d_i}{n}} D_{b_i}(T,I) = \sum_{\substack{i=1}}^{\overset{I \geq d_i}{n}} \left\lfloor \frac{I - r_i - d_i}{p_i} + 1 \right\rfloor c_i$$

If the demand bound function for each deadline is smaller then the maximal possible processor demand the task set is feasible:

$$\forall I : D_b(T,I) \leq I$$

For an example see figure 2 which is an output of function *plotDBf*. This test is called processor demand analysis and is implemented in function *DBftest*.

However, the computation of the demand bound function is complex. Because the function is discontinuous for each task instance at the deadline of the task the demand bound function must be calculated for each instance of every task separately. The total number of test points that have to be calculated depends on the task's periods and their deadlines. In the package the test points are generated in function *EDFtestPoints*.

2.4 Approximated test algorithms for EDF

In real life examples often tasks of the operating system have deadlines in terms of milli seconds while tasks of the application are executed in seconds. In such a case a large number of test points must be calculated when using the processor demand analysis test.

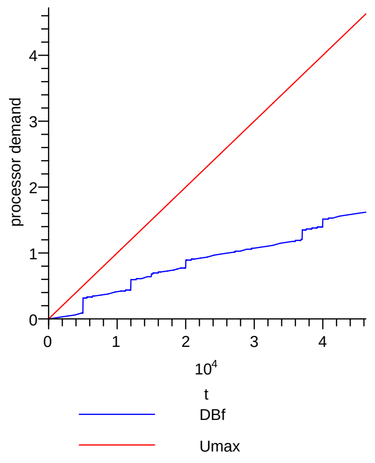

Figure 2: output of function “plotDBf”

In general the feasibility test is NP-hard. The computational complexity of the problem leads to approximative feasibility algorithms. These algorithms calculate results in linear time. The price for this is a smaller accuracy of the result. A good overview to the problems of approximative real-time analysis is given by Albers and Slomka [6], [7].

2.4.1 Chakraborty’s approximation

One solution to the problem was given by Chakraborty et al. [5]. To reduce the number of test points the maximum test interval

$$I_{max} = \frac{U}{1-U} \cdot \max_{\forall i}(p_i - D_i)$$

is divided into k equal test intervals. Only at the borders of these intervals the demand bound function is computed. The processor demand between the borders is summed up and brought forward in time to the previous border. So only k test points have to be computed. As figure 3 (output of function *plotChakrabortyDBf*) shows, the approximated demand bound function is always equal or greater than the original demand bound function. The approximated demand bound function is implemented in function *chakrabortyDBf*, the computation of the set of test points in *chakrabortyTestPoints* and the approximated processor demand test in *chakrabortyDBfTest*.

2.4.2 Approximation by superposition

An other approach to an approximated processor demand analysis is the approximation by superposition introduced in [7]. The idea of the algorithm is

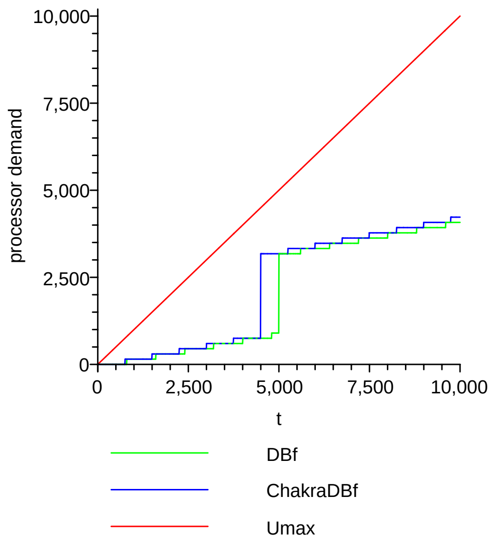

Figure 3: output of function "plotChakrabortyDBf"

to limit the number of test points separately for each task by constructing an approximated demand bound function for each task and to superpose then all approximations to a single approximated demand bound function. The number of test points can be reduced easily by linearization of the demand bound function. However, only the linearization of the demand bound function leads to a bad approximation. A better approximation can be constructed by computing exactly a firm number of k test points and to approximate all left test points by linearization. If $k+1$ test points are considered the maximal test interval for each task is given by

$$I_m(\tau_i) = k \cdot p_i + r_i + d_i$$

and the approximated demand bound function for a single task using I_m and $D_{b_i}(T, I)$ is given by

$$D'_{b_i}(T, I) = \begin{cases} D_{b_i}(T, I_m(\tau_i)) + \frac{c_i}{p_i} \cdot (I - I_m(\tau_i))), & I > I_m(\tau_i) \\ D_{b_i}(T, I), & I \leq I_m(\tau_i) \end{cases}$$

The approximated demand bound function for the task set is defined by

$$D'_b(T, I) = \sum_{i=1}^{n} D'_{b_i}(T, I)$$

As figure 4 (output of function *plotSuperpositionDBf*) shows, the approximated demand bound function is always equal or greater than the original demand bound function.

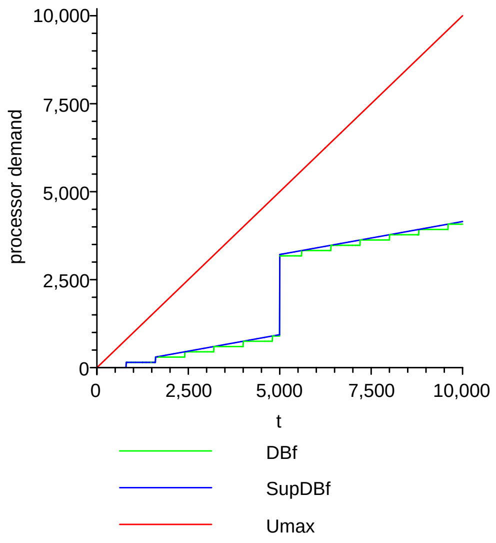

Figure 4: output of function “plotSuperpositionDBf”

The approximated demand bound function using superposition is implemented in function *superpositionDBf*, the computation of the set of test points in *superpositionTestPoints* and the approximated processor demand test in *superpositionDBfTest*. In addition a test for superposition is implemented using a dynamic error test in function *superpositionDynamicErrorTest*.

2.5 User interface functions using Maplets

The sub-package “tools” uses many functions from the package “realTimeAnalysis” to offer the user interactive graphical functions. They are used to help the user to easy start working with the package. It contains: A method for the interactive creation of tasks and task sets (*createTaskset*), a possibility to easy handle the plots of demand bound functions (*DBfPlotter*) and a method to give an overview about the different schedulability tests for a task set (*schedulabilityTests*).

2.6 Package overwiev

The selection of the functions and methods which have been implemented orients itself to the contents of the course “Real-time systems” teached at the University of Oldenburg. For usability of the package graphical user interfaces were implemented for suitable functions. Of course there are help files with explanations and examples for every function, that can be accessed through the Maple help-system.

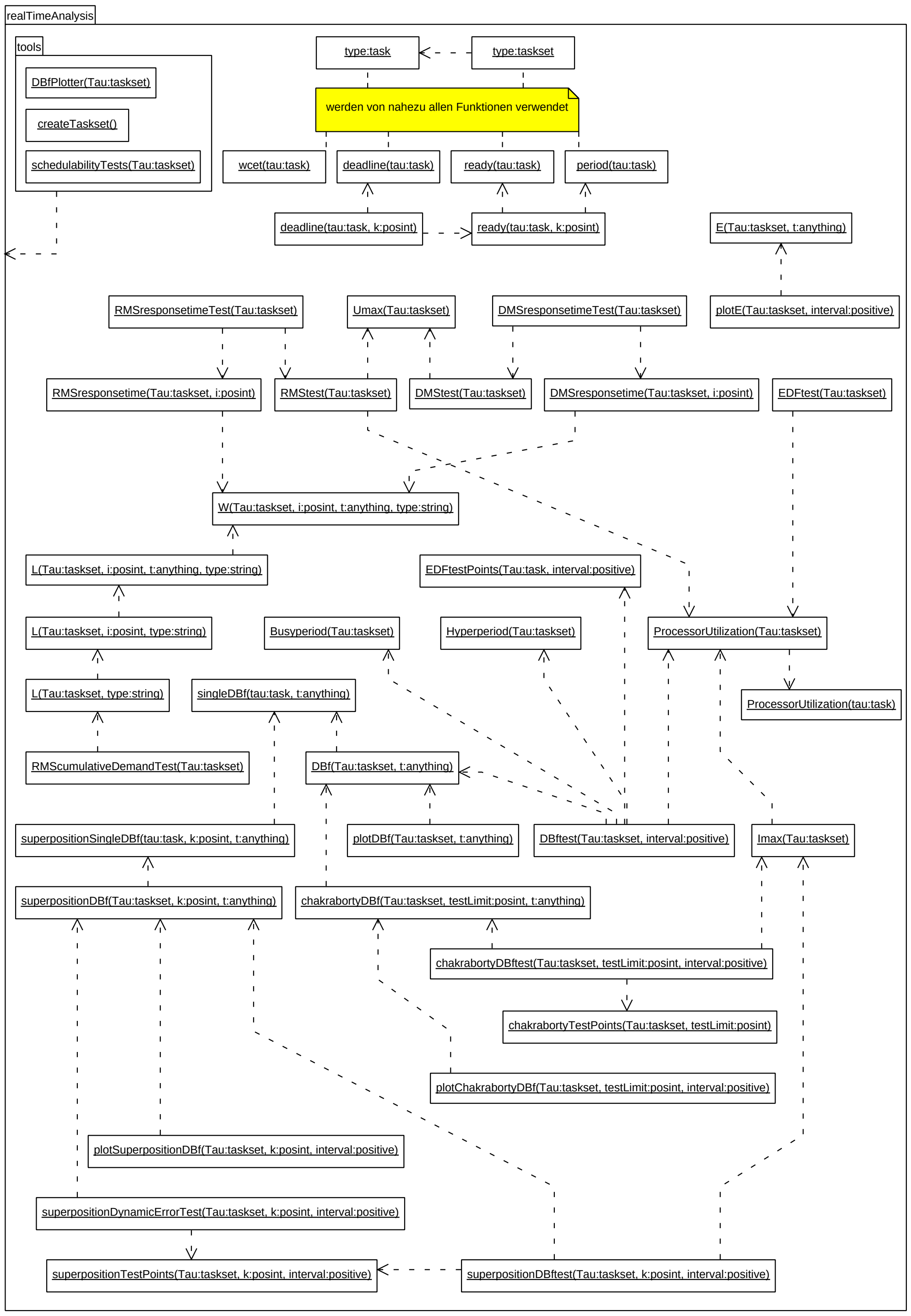
realTimeAnalysis
tools
DBfPlotter(Tau:taskset)
createTaskset()
schedulabilityTests(Tau:taskset)
type:task
type:taskset
werden von nahezu allen Funktionen verwendet
wcet(tau:task)
deadline(tau:task)
ready(tau:task)
period(tau:task)
deadline(tau:task, k:posint)
ready(tau:task, k:posint)
E(Tau:taskset, t:anything)
RMSresponsetimeTest(Tau:taskset)
Umax(Tau:taskset)
DMSresponsetimeTest(Tau:taskset)
plotE(Tau:taskset, interval:positive)
RMSresponsetime(Tau:taskset, i:posint)
RMStest(Tau:taskset)
DMStest(Tau:taskset)
DMSresponsetime(Tau:taskset, i:posint)
EDFtest(Tau:taskset)
W(Tau:taskset, i:posint, t:anything, type:string)
L(Tau:taskset, i:posint, t:anything, type:string)
EDFtestPoints(Tau:task, interval:positive)
L(Tau:taskset, i:posint, type:string)
Busyperiod(Tau:taskset)
Hyperperiod(Tau:taskset)
ProcessorUtilization(Tau:taskset)
L(Tau:taskset, type:string)
singleDBf(tau:task, t:anything)
ProcessorUtilization(tau:task)
RMScumulativeDemandTest(Tau:taskset)
DBf(Tau:taskset, t:anything)
superpositionSingleDBf(tau:task, k:posint, t:anything)
plotDBf(Tau:taskset, t:anything)
DBftest(Tau:taskset, interval:positive)
Imax(Tau:taskset)
superpositionDBf(Tau:taskset, k:posint, t:anything)
chakrabortyDBf(Tau:taskset, testLimit:posint, t:anything)
chakrabortyDBftest(Tau:taskset, testLimit:posint, interval:positive)
chakrabortyTestPoints(Tau:taskset, testLimit:posint)
plotChakrabortyDBf(Tau:taskset, testLimit:posint, interval:positive)
plotSuperpositionDBf(Tau:taskset, k:posint, interval:positive)
superpositionDynamicErrorTest(Tau:taskset, k:posint, interval:positive)
superpositionTestPoints(Tau:taskset, k:posint, interval:positive)
superpositionDBftest(Tau:taskset, k:posint, interval:positive)

Figure 5: Structure of the package

A Maple package is a collection of equal functions, but of course there are dependencies between the implemented functions in this package. Figure 5 gives an overview about the implemented functions and the dependencies between them using a in form of a dependency graph with a syntax based on UML.

3 Example application of the package

As an example application the task set of a hypothetical flight control system developed by Tindell and Clark [10] is used to show the functionality of some functions of the package. The definition of the previously made out task set T in Maple is easy:

```
> Tau:=[[800    , 0, 800    , 150 ],
        [200000 , 0, 5000   , 2277],
        [40000  , 0, 15000  , 420 ],
        [20000  , 0, 20000  , 552 ],
        [20000  , 0, 20000  , 496 ],
        [25000  , 0, 12000  , 1423],
        [50000  , 0, 50000  , 3096],
        [59000  , 0, 59000  , 7880],
        [50000  , 0, 100000 , 1996],
        [100000 , 0, 100000 , 3220],
        [100000 , 0, 100000 , 3220],
        [200000 , 0, 100000 , 520 ],
        [200000 , 0, 200000 , 1120],
        [1000000, 0, 200000 , 954 ],
        [200000 , 0, 200000 , 1124],
        [200000 , 0, 200000 , 3345],
        [1000000, 0, 1000000, 1990]];
```

with `[p, r, d, c]`.

After loading the package with “`with(realTimeAnalysis)`” all test functions for the different scheduling algorithms are ready to use:

3.1 Tests for RMS and DMS

For the RMS and DMS scheduling algorithms the test using the processor utilization (computed by the function U) can be used to analyze the feasibility of the task set:

```
> U(Tau);
19233803
--------
29500000

> DMStest(Tau);
maybe

> RMStest(Tau);
true
```

It can be seen that the task set can be scheduled with RMS, but for DMS this test is not accurate. To solve the problem a different test method using the worst-case response time of the tasks is used:

```
> DMSresponsetimeTest(Tau);
true

> RMSresponsetimeTest(Tau);
true
```

This tells us that the task set can be scheduled with both algorithms, RMS and DMS. The next command shows the worst-case response time of every task for DMS and the belonging deadline, which must be greater for every task to be schedulable:

```
> seq(
   [DMSresponsetime(Tau,i),d(Tau[i])],
   i=1..nops(Tau)
  );
[150, 800], [2877, 5000], [5170, 15000],
[5872, 20000], [6368, 20000], [4600, 12000],
[10214, 50000], [19894, 59000], [23688, 100000],
[29381, 100000], [33351, 100000], [34021, 100000],
[35441, 200000], [36545, 200000], [37969, 200000],
[43832, 200000], [46272, 1000000]
```

3.2 Tests for EDF

For the EDF scheduling algorithm 2 different test methods are implemented, the first one using the processor utilization and the second one performing the processor demand analysis by using the demand bound function:

```
> EDFtest(Tau);
true

> DBftest(Tau);
true
```

The test by processor demand analysis can also be displayed graphically:

```
> plotDBf(Tau);
```

For the Maple output of the function see figure 2.

3.3 Approximative tests for EDF

The exact computation of the EDF processor demand test is very slow because of the many test points that have to be evaluated. To get the results faster two approximative processor demand tests are implemented, Chakraborty's approximation of the demand bound function and the approximation by superposition:

```
> chakrabortyDBftest(Tau, 2000);
true

> superpositionDBfTest(Tau, 1);
true

> superpositionDynamicErrorTest(Tau, 10);
true
```

To show the results graphically also plotting functions are available for Chakraborty's approximation (figure 3) and approximation by superposition (figure 4):

```
> plotChakrabortyDBf(Tau, 2000, 10000);

> plotSuperpositionDBf(Tau, 1, 10000);
```

4 Efficiency

An important characteristic characteristic of the implementation that should be seen is the speed of the computations. In order to ensure that the results are computed as fast as possible the methods and functions have been tested with the Maple profiling tools. They provide functions to measure the memory use as well as the time used for computation. The speedup of many function could be achieved by using some other Maple constructs and methods that are equal to well known programming constructs.

E.g. in the function that computes the test points for the EDF feasibility test by processor demand analysis the computation can be done in a *while* or a *for*-loop, which is very slow. A better way is to use special Maple methods to dynamically create lists, like *seq*. Both variants can be seen in the following two functions which produce the same results:

-
  ```
  EDFtestPoints1 := proc(Tau::taskset, interval::positive)::set;
  local k::nonnegint, i::posint, t::nonnegative,
        D1::set, D2::set;
    k := 0; t := 0; D1:={}; D2:={};
    while (true) do
      for i from 1 to nops(Tau) do
        if ((p(Tau[i]) = infinity)) then
          t := d(Tau[i]) + r(Tau[i]);
        else
          t := k * p(Tau[i]) + d(Tau[i]) + r(Tau[i]);
        end if;
        if (t <= interval) then
          D1 := D1 union {t};
        end if;
      end do;
      if (D1 = D2) then
        return D1;
      else
        D2 := D1;
      end if;
      k := k + 1;
    end do;
  end proc;
  ```

-
  ```
  EDFtestPoints2 := proc(Tau::taskset, interval::positive)::set;
  local t::nonnegint, tau::task, D::set;
    D := {};
    for tau in Tau do
      if (p(tau) = infinity) then
        D := D union {d(tau) + r(tau)};
      else
        D := D union {seq(d(tau, t),
          t = 1..(floor( (interval-d(tau)-r(tau))/p(tau) ))+1)};
      end if;
    end do;
  end proc;
  ```

Using the profiling tools it can be seen, that the loop variant uses 9-times more time and memory than the version that builds the lists dynamically:

-
  ```
  EDFtestPoints1 := proc(Tau::taskset, interval::positive)
  local k::nonnegint, i::posint, t::nonnegative,
        D1::set, D2::set;
       |Calls Seconds  Words|
  PROC |    1   3,616 15960691|
     1 |    1   0,000      0| k := 0;
     2 |    1   0,000      0| t := 0;
     3 |    1   0,000      0| D1 := {};
     4 |    1   0,000      0| D2 := {};
     5 |    1   0,000      5| do
     6 | 1251   0,010   3753|   for i to nops(Tau) do
     7 |21267   0,693 2658412|     if p(Tau[i]) = infinity then
     8 |    0   0,000      0|       t := [...]
                                 else
     9 |21267   2,813 12191232|       t := k*[...]
                                 end if;
    10 |21267   0,040  88740|     if t <= interval then
    11 | 1517   0,040 1012843|       D1 := `union`(D1,{t})
                                 end if
                               end do;
    12 | 1251   0,020   5706|   if D1 = D2 then
    13 |    1   0,000      0|     return D1
                               else
    14 | 1250   0,000      0|     D2 := D1
                               end if;
    15 | 1250   0,000      0|   k := k+1
                             end do
  end proc
  ```

-
  ```
  EDFtestPoints2 := proc(Tau::taskset, interval::positive)
  local t::nonnegint, tau::task, D::set;
       |Calls Seconds  Words|
  PROC |    1   0,400 1914464|
     1 |    1   0,000      0| D := {};
     2 |    1   0,000      0| for tau in Tau do
     3 |   17   0,000   2125|   if p(tau) = infinity then
     4 |    0   0,000      0|     D := `union`([...])
                               else
     5 |   17   0,400 1912339|     D := `union`([...])
                               end if
                             end do
  end proc
  ```

Further optimizations of the factor 3 - 4 can be achieved by using the function *add* instead of *sum*, which is appropriate for these functions.

These examples are also detailed shown in file *IPprofile.mw*.

5 Conclusion

The package "realTimeAnalysis" offers many basic functions and test methods for different scheduling algorithms for real-time systems. In addition functions with graphical user interfaces and detailed help files are offered, which simplify the use especially for Maple beginners. So this package can help teaching the basic concepts of real-time system analysis and understanding the mathematic principles behind them.

The package is logically arranged by using subpackages and can be easily extended with further packages for new functions and data types. To simplify that the source code will be made available under GNU General Public License and can be downloaded fom the internet.

A Source code

The sources of the package and the pre-compiled binaries for use with Maple 9.5 and Maple 10 can be downloaded at:

`http://www.mariokorte.de/downloads/easytrack.php?id=ip1`.

References

[1] C. L. Liu and James W. Layland *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment* Journal of the ACM, pages 46 - 61, 1973

[2] A. Burns and A. Wellings *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems* Elsevier, Oxford, 1995

[3] S. Baruah, A. Mok and L. Rosier *Preemtive Scheduling Hard-Real-Time Sporadic Tasks on One Processor* Proceedings of the Real-Time Systems Symposium, pages 182 - 190, 1990

[4] K. Gresser *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme* PhD Thesis, Lehrstuhl für Prozessrechner, Technical University of Munich. Fortschrittsberichte VDI, Reihe 10, Nr. 268, VDI Verlag, Duesseldorf, 1993

[5] Samarjit Chakraborty, Simon Künzli and Lothar Thiele *Approximate Schedulability Analysis* 23rd IEEE Real-Time Systems Symposium (RTSS), IEEE Press, pages 159 - 168, 2002

[6] Karsten Albers and Frank Slomka *Efficient Feasibility Analysis for Real-Time Systems with EDF scheduling* IEEE Procedings of the DATE (Design, Automation and Test in Europe) Conference 2005, pages 492 - 497, 2005

[7] Karsten Albers and Frank Slomka *An Event Stream Driven Approximation for the Analysis of Real-Time Systems* IEEE Procedings of the 16th Euromicro Conference On Real-Time Systems, 2004

[8] M. Joseph and P. Panday *Finding Response Times in Real-Time Systems* The Computer Journal(29/5), pages 390 - 395, 1986

[9] John Lehoczky, Lui Sha and Ye Ding *The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior* CH2803-5/89/0000/0166/$01.00, 1989

[10] Ken Tindell and John Clark *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems* Microprocessing and Microcomputing 50, pages: 117 - 134, 1994

[11] Giorgio C. Buttazzo *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* Kluwer Academic Publishers, 2002.