Deadline-Monotonic Software Scheduling for the Co-Synthesis of Parallel Hard Real-Time Systems

Peter Altenbernd
CADLAB, D-33094 Paderborn – Germany
Tel.: ++49/5251/284-123, Fax.: ++49/5251/284-140, Email: peter@cadlab.de

Abstract

This paper focuses on software scheduling in hard real-time embedded systems. It uses the deadline-monotonic scheduling heuristics, where the analysis whether the hard real-time conditions are met, is done by a schedulability test. The test presented in this paper overcomes the problems of existing approaches with parallel communicating tasks. The essential of the test is, that the communication caused precedence constraints are mapped to minimum-maximum offset intervals, to deal with multiperiod systems, where fixed offset values are insufficient. By this method the deadlines and the offset intervals are computed automatically during the efficient analysis.

1 Introduction

Recently, software synthesis becomes more and more important in the automation of embedded system design. An embedded system is a special purpose computer consisting of one or more controllers. Low-cost embedded systems can be constructed by using microcontrollers or core processors that do not use operating systems. Many embedded systems must meet hard real-time conditions, in order to avoid catastrophic failures. The software for hard real-time systems is often designed by a number of periodic tasks in a parallel environment.

In this paper the scheduling of hard real-time systems is addressed, at which the following assumptions are made:

- Each task re-arrives periodically.
- For each arrival, a task executes a bounded amount of computation.
- Each task must meet a hard real-time condition, termed the *deadline*, measured relative to the beginning of the period of the task, i.e. the task must be guaranteed by the schedule to have finished its computation before this deadline.
- Tasks may receive data from one or more other tasks at their beginning, and tasks may send data to one or more other tasks at their end.
- Communicating tasks share the same period length.
- The parallel communicating task set is allocated on a number of identical processors.
- Scheduling is done locally for each processor.

Static priority preemptive scheduling methods are an efficient way of constructing and analyzing schedules for hard real-time systems. Priority preemptive based scheduling heuristics dispatch tasks in priority order, and the priorities remain static during runtime. Lower priority tasks can be suspended by higher priority tasks. Allowing task deadlines to be before the end of the periods, and assigning each task a unique priority with respect to its deadline (the smaller the deadline, the higher the priority) is called *deadline-monotonic scheduling (DMS)* [6]. DMS is very well-suited for parallel environments with communicating task, since the effort following a sending task can be taken into consideration by reducing its deadline, and hence increasing its priority.

Static priority preemptive scheduling is not a correct-by-construction approach with respect to guarantee the task deadlines. In order to predict whether the deadlines are met, *schedulability tests* are applied (first presented by Liu and Layland [7] in 1973). In a schedulability test the *worst-case response time* of each task is determined, which is the longest time ever taken by the task from the beginning of its period until it completes its required computation, i.e. including possible interferences by higher priority tasks. If the worst-case response times of all tasks are less or equal to their corresponding deadlines, all hard real-time conditions are guaranteed to be met. Due to a better runtime complexity schedulability tests are preferred to simulation. In a simulation the least-common multiple of the task periods must be inquired, which can last very long. Most schedulability tests are *not-necessary and sufficient*, i.e. they do not deliver an exact result (to further reduce runtime) but the computed worst-case response time are guaranteed to be not too optimistic.

Schedulability tests for communicating tasks in parallel environments are still a difficult problem. In such systems precedence constraints may arise, which existing schedulability tests try to handle by fixed *offset*

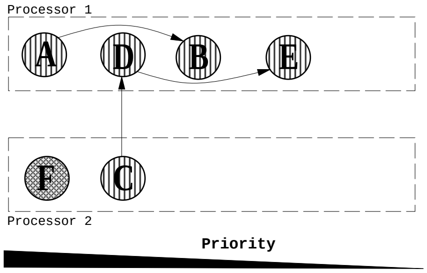

Figure 1: Precedence graph of a communicating task set

values. An offset value represents the arrival of a certain task with respect to the beginning of the period. The representation of precedence constraints as *fixed* offsets is often insufficient in a system of tasks with different periods, since the arrival time of a receiving task may vary due to the interference caused to its predecessors. In the example of Fig.1 a communicating task set is shown: The nodes of the graph represent the tasks, the edges represent communications. In the example it is assumed that the period of task F is different from the common period of the other tasks, and that the priority is decreasing from left to right in the picture. Due to a communication, Task D starts when C is finished. Due to the higher priority of F, C may be interfered by F, or may not. This results in two different arrival times of D, a *minimum* and a *maximum offset*. Checking task D for schedulability by assuming the minimum offset of D, the test would be too optimistic, since it is too easy to meet the deadline of D. Taking the maximum value instead overcomes this problem. But checking task B for schedulability, the maximum offset of D may lead to underestimate the interference that D can cause to B (due to the higher priority of D) resulting again in a too optimistic view.

Hence, this paper presents a not-necessary and sufficient DMS schedulability test for parallel communicating tasks that handles minimum and maximum offsets. For testing the current task on schedulability the latest possible arrival time, the maximum offset, is assumed. For a task that may interfere the current task due to a higher priority, an arrival in between the minimum and maximum value is assumed. The computed response time of the current task is used to determine the min/max offsets of its successors. Because of its low runtime complexity, the test is very well-suited to be used as a subroutine of task-to-processor allocation. In distinction to former approaches the designer does not need to care about appropriate deadlines and offsets, which are computed automatically by this method.

The remainder of the paper is structured as follows. The next section illustrates how this approach is related to others. In Section 3, the new schedulability test is described. The mapping towards DMS (i.e. the deadline computation) is done in Section 3.1, and the actual test is presented in Section 3.2. In Section 3.3, an example is discussed, and experimental results are presented. Finally, Section 4 gives some conclusions.

2 Related Work

If precedence constraints are represented by *fixed* offsets, schedulability testing may lead to dangerous optimistic results (see above). Nevertheless offsets are useful for other situations like mutual exclusion.

Schedulability tests which ignore offsets, like [1], assume that all tasks on a processor arrive simultaneously. Since this is not true for the sequential behavior of communicating tasks, the test may overestimate some worst-case response times. There is the possibility to make the test less pessimistic by modifying the task set parameters (like in [3]) in order to pass the test. In contrast to this, including offsets directly into the analysis allows evaluating the test results without re-translating the modified task set into the initial one.

The most accurate analysis which includes fixed offsets is presented by Tindell [9]. For this purpose, the open framework of the University of York [3] is used, which is appropriate to simultaneously consider many aspects of schedulability analysis. In contrast to Tindell's approach, the method presented in this paper overcomes the above mentioned fixed-offsets problem by handling min/max offsets, and determines offsets and deadlines automatically. Many of the features of Tindell's test, like the arbitrary deadlines, are not needed for DMS schedulability analysis. Because of this, this paper focuses on the essential needs of scheduling parallel communicating tasks by the DMS approach.

The problem with correct-by-construction approaches (like [5]), when determinating a feasible schedule of systems involving different periods, is that the least-common multiple (LCM) of the periods must be inquired. If this is combined with allocation (NP-hard) [11], it is still harder (impossible for large systems and large LCMs) to determine the schedule. However, in this paper the well-known, optimal and more flexible DMS heuristics (i.e. not NP-hard) is used, that allows efficient schedulability testing which at most needs to inquire the length of the longest period.

Another correct-by-construction approach is, to find *static cyclic schedules* [12], by constructing a set of tasks sharing the same period length. As pointed out in [2] this procedure is very similar to the approach of modifying the task set. Hence, static cyclic scheduling is merely a special case of a schedulability algorithm that includes offsets.

Using fixed priority scheduling heuristics in an overall manner and using the allocation as a subroutine (e.g. [4]), it is difficult to address the allocation problem adequately, resulting in poor processor utilizations. Utilization can be enhanced by handling the allocation at the highest level (like in [10]), and by performing scheduling (incl. test) at a lower level. This is supported by the method of this paper and in contrast to [10] the precedence constraints are taken into consideration.

3 The Schedulability Test

Before the actual schedulability test, the precedence constraints are mapped to the deadline-monotonic approach (see Section 3.1). After that as the main part of the test, the worst-case response time is computed for each task (see Section 3.2), taking into account the former mapping and the precedence constraints.

The used notations are:

T_p = The period of task p

$C_p^{min/max}$ = The minimum/maximum computation time of task p

D_p = The static deadline of task p, measured relative to the beginning of the period of the task, and $D_p \leq T_p$

h_p = The processor where task p is hosted

The tasks compose a directed acyclic *precedence graph* $PG = (V, E)$ where V is the set of tasks, and E is the set of edges representing communications between tasks, i.e. if task a sends to task b: $a \rightarrow b \in E$.

3.1 Mapping the Precedence Graph to DMS

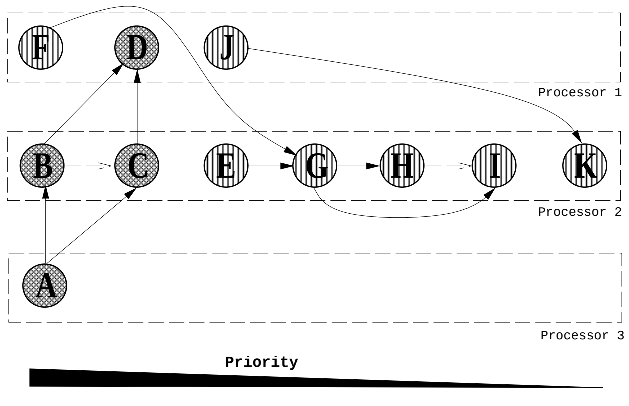

Figure 2: Precedence graph of a communicating task set

The mapping towards DMS is done by computing appropriate deadlines using the precedence graph. The deadline, D_p, does not take into account the computations following p. Furthermore, it cannot be assumed that all tasks are dispatched in parallel. If a task A sends to tasks B and C, and B and C are hosted on the same processor, B and C are always processed consecutively (see Fig.2, the non-dashed edges only). This means that for one of the two tasks, either B or C, the deadline is too optimistic. To avoid this, further edges expressing the sequential behavior (e.g. $B \rightarrow C$, the dashed edges in Fig.2) are added to the precedence graph resulting in the definition of an *extended precedence graph* $EPG = (V, E^{'})$ with:

$$E(0) := E$$

$$\begin{aligned} &\forall : 0 \leq i < m : \\ &\quad E(i+1) := \\ &\qquad E(i) \cup \{ q_1 \rightarrow q_2 \mid \exists p : \; p \rightarrow q_1 \in E(i) \wedge \\ &\qquad\qquad p \rightarrow q_2 \in E(i), \\ &\qquad\qquad h_{q_1} = h_{q_2},\; d_{q_1}(i) \leq d_{q_2}(i) \; \} \end{aligned}$$

$$E^{'} := E(m)$$

If, for instance, task q_1 is followed by some subsequent tasks, while q_2 is the end of the line, deadline-monotonic scheduling is more effective by giving q_1 a higher priority. This fact is taken into consideration by the condition $d_{q_1}(i) \leq d_{q_2}(i)$ when determinating $E(i+1)$. (By further similar conditions it is possible to get a unique priority for each task). The extended precedence graph is used to compute the *real-deadline* values, d, which involve consecutive processing of parallel tasks on the same processor, and the effort following a task.

$$\begin{aligned} &\forall : 0 \leq i \leq m : \\ &\quad d_p(i) := \begin{cases} D_p, & \text{if } \nexists q : p \rightarrow q \in E(i) \\ MIN\{d_q(i) - C_q^{max} \mid p \rightarrow q \in E(i)\}, & \text{else} \end{cases} \end{aligned}$$

$$d_p := d_p(m)$$

The task set is mapped to DMS with respect to the real-deadline values, which are exploited by the schedulability test (see Section 3.2). With the help of the additional edges, it can easily be determined where a timing violation is located. Without these edges, the true error location could be hidden. Regarding the example of Fig.2, the schedulability test could report a timing violation with respect to B or C, but not with respect to A, which may be the real cause of the error. Recursively adding a new edge if the last added edge has imposed again the same situation, as expressed by the definitions of the $E(i)$, helps the schedulability test to accurately estimate the response time of tasks belonging to the same subgraph (see Section 3.2).

The computation of the real-deadlines can be performed by a breath-first-search (BFS) technique for each $E(i)$. Its runtime grows linear with the number of edges and vertices.

3.2 Worst Case Response Time

The actual schedulability test consists mainly of the computation of the *worst-case response time* of each task, r_p. For this purpose, it is distinguished between tasks with the same period, and tasks with distinct periods. According to this, a *transaction* is defined as the maximum set of vertices (tasks) of E' sharing the same period length. The distinction is done to appropriately handle the offset intervals.

The *worst-case response time* is defined as:

$$r_p := I_p + r_p^{Tmax}$$

where I_p denotes the *interference*, and r_p^{Tmax} the *maximum transaction response time*. The worst-case response time must be compared to the deadline, i.e. if $r_p \leq d_p$, p is guaranteed to hold its deadline.

The interference is given only with respect to other transactions, as the next definition states.

$$I_p := \sum_{q \in hp(p)} \left\lceil \frac{d_p - o_p^T}{T_q} \right\rceil C_q^{max}$$

The *higher priority set* $hp(p) = \{t_1, ..., t_m\}$ is the set of higher priority tasks of p with $\forall : 1 \leq i \leq m : T_{t_i} \neq T_p$, and $h_{t_i} = h_p$.

Thus, the interference value gives the amount of time a task is prevented from finishing by other tasks. The interference might happen between o_p^T, called the *transaction offset*, and d_p. o_p^T is equal to the beginning of the earliest higher priority task of the transaction on the processor, as the next definition states.

$$\begin{aligned} o_p^T := MIN\{o_q^{min} \mid h_p = h_q \ \wedge T_q = T_p \\ \wedge\, q \in hp^T(p) \cup \{p\}\ \} \end{aligned}$$

(See below for the definition of minimum offset, o^{min}.)

The definition of the *transaction higher priority set* $hp^T(p)$ needed above is as follows:
$hp^T(p) = \{q_1, ..., q_n\}$ is the set of higher priority tasks of p with $\forall : 1 \leq i \leq n : T_{q_i} = T_p$ and $h_{q_i} = h_p$.

The rest of this section deals with the handling of the interferences caused by tasks of the same transaction. It is distinguished between the earliest possible arrival of a task, the *minimum offset*, o^{min}, and the latest possible arrival of a task, the *maximum offset*, o^{max}. Regarding a task p, the maximum-offset, o_p^{max}, represents that the beginning of p is deferred as much as possible towards the end of the period, which makes it harder to meet the deadline of p. The *minimum offset*, o_q^{min}, is of interest for the interference a higher priority task q of the same transition can cause to p. The interval $[o_q^{min}, o_q^{max}]$, is the interval in which q arrives, and hence can interfere p. The two offset types are defined as follows.

$$o_p^{max} := \begin{cases} 0, & \text{if } \nexists q : q \rightarrow p \in E' \\ MAX\{o_{pq}^{max} \mid q \rightarrow p \in E'\}, & \text{else} \end{cases}$$

with

$$o_{pq}^{max} := \begin{cases} r_q^{Tmax}, & \text{if } h_q = h_p \\ r_q, & \text{else} \end{cases}$$

and

$$o_p^{min} := \begin{cases} 0, & \text{if } \nexists q : q \rightarrow p \in E' \\ MAX\{r_q^{Tmin} \mid q \rightarrow p \in E'\}, & \text{else} \end{cases}$$

In other words, the maximum offset of a task is either 0 if there is no predecessor, or equal to the *maximum transaction response time* (the maximum response time of a task with respect to its transaction; see below) of a predecessor hosted on the same processor, or equal to the worst-case response time of a predecessor hosted on a distinct processor. The minimum offset of a task is either 0 if there is no predecessor, or equal to the *minimum transaction response time* (the minimum response time of a task with respect to its transaction; see below) of a predecessor.

The transaction response times, $r_p^{T(min/max)}$, are computed without taking into account the interference of tasks outside the transaction, i.e. $r_p^{T(min/max)} \leq r_p$. The definition of the *minimum (maximum) transaction response time* is as follows.

$$r_p^{T(min/max)} := o_p^{(min/max)} + I_p^{T(min/max)} + C_p^{(min/max)}$$

For this formula the *minimum (maximum) transaction interference*, $I^{T(min/max)}$, is needed which is the minimum (maximum) interference caused by tasks of the same transaction (see below).

In the following definitions, four different situations are distinguished. First, the considered task p is dispatched earlier than a higher priority task q arrives, (denoted as **(early)** in the following definitions). Second, p arrives later than q is dispatched (denoted as **(late)**). In these two situations, p is not interfered by q. Third, p arrives while q is running (denoted as **(wait)**). In this case, p is interfered by the remaining part of q. And last, p is preempted by q (denoted as **(pree)**). In this case, p is interfered by the full computation time of q.

In each of the four cases, the arrivals of q and p are assumed in a way to keep the minimum (maximum) transaction interference as low (high) as possible. Nevertheless, the minimum transaction interference does not need to be less or equal to the corresponding maximum value. In the case of an extreme early arrival of a task the interference may be higher than in the case of a later arrival. Only the minimum transaction response time is always less or equal to the corresponding maximum value. The transaction interference values are defined as follows.

$I_p^{Tmin}(0) := 0$

$$\begin{array}{l}
\forall : 1 \leq i \leq n : \\
\quad I_p^{Tmin}(i) := I_p^{Tmin}(i-1) \\
\quad + \begin{cases}
r_{q_i}^{Tmin} - s_p^{min}(i-1), & \textbf{(wait)} \\
\quad \text{if} \quad s_{q_i}^{min} \leq s_p^{min}(i-1) < r_{q_i}^{Tmin} & \\
\qquad \wedge\, s_{q_i}^{max} < o_p^{min} + I_p^{Tmin}(i-1) + C_p^{min} & \\
C_{q_i}^{min}, & \textbf{(pree)} \\
\quad \text{if} \quad s_p^{min}(i-1) < s_{q_i}^{min} & \\
\qquad \wedge\, \underline{s_{q_i}^{max} < o_p^{min} + I_p^{Tmin}}(i-1) + C_p^{min} & \\
\qquad \wedge\, s_{q_i}^{min} \leq o_p^{min} < r_{q_i}^{Tmin} & \\
0, & \textbf{(early, late)} \\
\quad \text{else} &
\end{cases}
\end{array}$$

$I_p^{Tmin} := I_p^{Tmin}(n)$

with: $q_i \in hp^T(p)$ and $\forall 1 \leq i < n : \; s_{q_i}^{min} \leq s_{q_{i+1}}^{min}$

$I_p^{Tmax}(0) := 0$

$$\begin{array}{l}
\forall : 1 \leq i \leq n : \\
\quad I_p^{Tmax}(i) := I_p^{Tmax}(i-1) \\
\quad + \begin{cases}
r_{q_i}^{Tmax} - s_p^{max}(i-1) & \textbf{(wait)} \\
\quad \text{if} \quad s_{q_i}^{max} \leq s_p^{max}(i-1) < r_{q_i}^{Tmax} & \\
C_{q_i}^{max} & \textbf{(pree)} \\
\quad \text{if} \quad s_p^{max}(i-1) < s_{q_i}^{max} \,\wedge & \\
\qquad o_{q_i}^{min} \underline{< o_p^{max} + I_p + I_p^{Tmax}}(i-1) + C_p^{max} & \\
\qquad \wedge\, s_{q_i}^{max} \leq o_p^{max} < r_{q_i}^{Tmax} & \\
0 & \textbf{(early, late)} \\
\quad \text{else} &
\end{cases}
\end{array}$$

$I_p^{Tmax} := I_p^{Tmax}(n)$

with: $q_i \in hp^T(p)$ and $\forall 1 \leq i < n : \; s_{q_i}^{max} \leq s_{q_{i+1}}^{max}$

The negated expressions in these formulas are not necessary to express the **(pree)** condition, but to avoid that a q_i is taken into account twice. The $I_p^{T(min/max)}(i)$-values are determined in increasing order over the *min/max start-time* values, $s^{min/max}$, for each q_i. The min/max start time declares the earliest/latest possible time instant when a task starts processing with respect to the transaction. The start-times are computed simultaneously with the transaction interferences, as expressed by the next definitions.

$s_p^{min}(0) := o_p^{min}$

$$\begin{array}{l}
\forall : 1 \leq i \leq n : \\
\quad s_p^{min}(i) := \begin{cases}
r_{q_i}^{Tmin}, & \textbf{(wait)} \\
\quad \text{if} \quad s_{q_i}^{min} \leq s_p^{min}(i-1) < r_{q_i}^{Tmin} & \\
\qquad \wedge\, s_{q_i}^{max} < o_p^{min} + I_p^{Tmin} + C_p^{min} & \\
s_p^{min}(i-1), & \text{else}
\end{cases}
\end{array}$$

$s_p^{min} := s_p^{min}(n)$

$s_p^{max}(0) := o_p^{max}$

$$\begin{array}{l}
\forall : 1 \leq i \leq n : \\
\quad s_p^{max}(i) := \begin{cases}
r_{q_i}^{Tmax}, & \textbf{(wait)} \\
\quad \text{if}\, s_{q_i}^{max} \leq s_p^{max}(i-1) < r_{q_i}^{Tmax} & \\
s_p^{max}(i-1), & \text{else}
\end{cases}
\end{array}$$

$s_p^{max} := s_p^{max}(n)$

Note, that the q_i and the value n are the same as in the definition of the min/max transaction interference. In the **wait** case, the start time of p is equal to the end of q_i. In all other cases, the real offset remains unchanged.

With the computation of the minimum and the maximum transaction interference, all components of the worst-case response time are determined, and can now be checked against the deadline.

The schedulability test can also be implemented by a BFS, at which each edge and each vertex is visited once. For computing the interference, I, there is an additional effort of at most $O(|V|)$ for each vertex. Hence, the complexity of the whole schedulability test is: $O(|E^{'}| + |V|^2)$.

3.3 Example and Experimental Results

In Fig.2 the precedence graph of a communicating task set is shown. The 11 tasks are located on three processors as indicated by the picture. The three subgraphs compose two transactions. It is assumed that all tasks have a constant computation time, C. More detailed information can be found in Fig.3.

Processor 1:

id	T	d	C	o	s	o^T	I^T	I	r^T	r
F	20	14	2	[0,0]	[0,0]	0	[0,0]	0	[2,2]	2
D	14	14	2	[6,6]	[6,6]	6	[0,0]	2	[8,8]	10
J	20	18	2	[0,0]	[2,2]	0	[2,2]	4	[4,4]	8

Processor 2:

id	T	d	C	o	s	o^T	I^T	I	r^T	r
B	14	10	2	[2,2]	[2,2]	2	[0,0]	0	[4,4]	4
C	14	12	2	[4,4]	[4,4]	2	[0,0]	0	[6,6]	6
E	20	14	3	[0,0]	[0,0]	0	[0,0]	4	[3,3]	7
G	20	16	2	[3,3]	[3,3]	0	[0,0]	8	[5,5]	13
H	20	18	2	[5,5]	[5,5]	0	[0,0]	8	[7,7]	15
I	20	20	2	[7,7]	[7,7]	0	[0,0]	8	[9,9]	17
K	20	20	2	[4,8]	[9,9]	0	[5,1]	8	[11,11]	19

Processor 3:

id	T	d	C	o	s	o^T	I^T	I	r^T	r
A	14	8	2	[0,0]	[0,0]	0	[0,0]	0	[2,2]	2

Figure 3: Input and output data of the example task set

The tasks are listed in priority order for each processor. It is assumed that the initial deadline, D, is equal to the period of each task. All information computed by the test is also included in the table. K is the only task with distinct minimum and maximum offsets. Nevertheless, the minimum transaction response time of K is equal to its maximum value.

The table in Fig.4 shows some experimental results about fixed examples which were analyzed by simulation and schedulability testing. The results show that

	n	m	t	LCM	sim	test	Q	Q^{FO}
tmn	11	3	2	140	0.5	<0.1	91	89
cpf	14	3	4	7920	4.5	<0.1	100	91
tnn	43	8	4	420	1.1	0.1	84	75
bnn	48	8	1	2000	2.4	0.1	99	52
mc	100	16	4	420	2.1	0.2	89	79

n = # tasks, m = # processors, sim = CPU-time of simulation,
t = # transactions, test = CPU-time of schedulability test

Figure 4: Experimental results

the schedulability test is much more efficient than simulation, though the examples and the LCMs are quite small. The test quality, Q, was determined by:

$$1 - \sum_{p \in V} \frac{r_p - r_p^{SIM}}{d_q}$$

where r^{SIM} is the worst-case response time delivered by the simulation. With large LCMs the test becomes more efficient, and with less tasks and transactions the test becomes more accurate. Similary the quality of a test using fixed offsets, Q^{FO}, was computed, but in contrast to above each negative term $r_p - r_p^{SIM}$ was replaced by d_q, which extremely penalizes underestimations. For the fixed offset schedulability test [9], the offsets were determined corresponding to the structure of extended precedence graph, at which the offset, o, of a node is the maximum value $o_q + C_q$ of all predecessors q. Comparing the quality values shows that the new test significantly enhances the accuracy of the test results.

4 Conclusions and Future Work

Due to the low costs of microcontrollers or core-processors, the software synthesis for hard real-time embedded systems becomes an emerging field. This paper targets in the use of the DMS heuristics, by presenting a sufficient and not-necessary DMS schedulability test for parallel communicating tasks. The precedence-constrained tasks are automatically mapped towards DMS by computing appropriate deadlines. Multiperiods are handled by minimum and maximum offsets as the representation of the precedence constraints. For testing the current task the maximum offset is taken into account, while for computing the interference a task can cause, the whole interval between the minimum and maximum value is considered. The test is efficient to implement by a BFS.

The schedulability test will be used to analyze real-time systems specified by Petri nets (Cadlab) [8] and differential equations (University of Paderborn), which is currently under development.

Acknowledgement The author would like to thank Carsten Ditze, and Frank Buijs for some helpful comments on an earlier version of this paper.

References

[1] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. *Hard Real-Time Scheduling: The Deadline Monotonic Approach.* In *IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[2] N. C. Audsley, K. Tindell, and A. Burns. *The End of The Line for Static Cyclic Scheduling?* In *Proceedings of the 5th Euromicro Workshop on Real-time Systems*, 1993.

[3] A. Burns. *Preemptive Priority Based Scheduling: An Appropriate Engineering Approach.* Report YCS214, Department of Computer Science, University of York, 1993.

[4] B.-C. Cheng, A.D. Stoyenko, and T.J. Marlowe. *Least-Space-Time-First Scheduling Algorithm: A Policy for Complex Real-Time Tasks in Multiple Processor Systems.* In *Proceedings of the WRTP'94*, 1994.

[5] P. Chou and G. Borriello. *Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems.* In *Proceedings of the DAC'94*, 1994.

[6] J.Y.T. Leung and J. Whitehead. *On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. Performance Evaluation*, 2(4):237–250, December 1982.

[7] C.L. Liu and J.W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of ACM*, 20(1):46–61, 1973.

[8] F. J. Rammig. *System Level Design. Nato Advanced Study Institute*, 1993.

[9] K. Tindell. *Adding Time-Offsets To Schedulability Analysis.* Report YCS221, Department of Computer Science, University of York, 1994.

[10] K. Tindell, A. Burns, and A. Wellings. *Allocating Real-Time Tasks (An NP-Hard Problem made Easy). Journal of Real-Time Systems*, 4(2):145–165, 1992.

[11] J. Xu. *Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.

[12] J. Xu and D. L. Parnas. *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.