Imprecise computation technique to schedule AND/OR tasks with global end-to-end deadline in distributed real-time systems

Wafaa A. El-Haweet, Hanan H. El-Meligy and Ibrahim Abd El-Salam
Computer Science and Automatic Control Dept., Alexandria University, Alexandria, Egypt

In a real-time system, tasks must produce logically correct results by their deadlines. These tasks usually don't need to be executed in a sequential order. Rather, certain precedence constraints must be handled between them along with timing constraints to provide correct scheduling and consequently correct execution. In traditional precedence-constrained scheduling, a task is ready to execute when all of its direct predecessors are completed. Such task is called an *AND* task. In many applications, some tasks are ready to execute when one or more but not necessarily all of their direct predecessors are completed. These tasks are called *OR* tasks. The resultant system containing both *AND* & *OR* tasks is said to have *AND/OR* precedence constraints. Directed Acyclic Graphs *DAG*s are used to represent these systems. This paper introduces a method to combine both precedence constraints and timing constraints into the same scheduling problem. Timing constraints here are in the form of a global End-to-End Deadline *EED* between a start task and an end task of the *AND/OR* graph representing the task system. A proposed Imprecise Computation Technique *ICT* is introduced to maximize the solution quality within the available time. *ICT* is motivated by the fact that one can trade off precision with timeliness; it prevents the total processing time of the system from stretching over the *EED.* To examine and evaluate this proposed imprecise scheduling solution, a Random Graph Generator algorithm *RGG* is created to provide a wide range of random *DAG*s to be used in the simulation experiments.

تزايدت أهمية أنظمة الوقت الفعلي في عدد كبير من مجالات التطبيق، وتكون فيها المهمة بمثابة الوحدة التي يتم جدولتها و من ثم تنفيذها و يجب علي مجموعة المهام أن تحقق نتائج سليمة في حدود الفترة الزمنية المسموح بها. لا يتم تنفيذ المهام غالبا في ترتيب متسلسل، ولكن يجب مراعاة قيود الأسبقية بين المهام بالإضافة إلى القيود الزمنية لتحقيق صحة الجدولة وبالتالي صحة التنفيذ. في حالة الجدولة التقليدية المصحوبة بقيود الأسبقية، تكون المهمة جاهزة للتنفيذ عند انتهاء جميع المهام السابقة لها. فيطلق علي هذه المهمة اسم "مهمة و". ولكن في تطبيقات أخري عديدة ، تكون بعض المهام جاهزة للتنفيذ إذا انتهت إحدى أو بعض وليس جميع المهام السابقة لها من أداء عملها. و يسمي هذا النوع من المهام "مهمة أو" . نظام المهام الذي يمزج بين النوعين السابقين يطبق عليه قيود الأسبقية المسماة "و/أو" . وتستخدم الشبكات الموجهة غير المحتوية علي دوائر مغلقة في تمثيل مثل هذه النظم. تضمن البحث طريقة مثالية جديدة للدمج بين القيود الزمنية و قيود الأسبقية في نفس عملية الجدولة. ويتمثل القيد الزمني في هذه الحالة بالفترة الزمنية الكلية المحددة منذ بداية تنفيذ أول مهمة في الشبكة الممثلة للنظام إلي الانتهاء من عمل آخر مهمة فيها. و تم تطوير الطريقة التقليدية في جدولة مثل هذه النظم إلي طريقة حسابات تقريبية لجدولة نظام المهام السابق للحصول علي نتائج عالية الجودة في حدود الوقت المسموح به. إن الفكرة الرئيسية التي تعتمد عليها هذه الطريقة للحسابات التقريبية هي التضحية ببعض من الدقة في النتائج في سبيل إنجاز معظم المهام في خلال الفترة الزمنية الكلية المحددة. ولفحص و تقييم هذه الطريقة، قمنا بتصميم مولد عشوائي للشبكات لتوفير نطاق واسع من الشبكات الموجهة غير المحتوية علي دوائر مغلقة لاستخدامها في تجارب المحاكاة.

Keywords: *AND/OR* precedence constraints, Directed acyclic graphs, Imprecise computation, Dependability, Design-to-time real-time scheduling

1. Introduction

Real-time systems are defined as those systems in which correctness depends on the logical result of computation as well as on the time at which the results are produced. Most real-time systems are designed using *ad hoc* techniques [1,2], making them very brittle, difficult to understand and expensive to modify. The primary reason behind this situation is that task representation is not always taken into consideration. A general model to represent a real-time task system can be in the form of *AND/OR DAG*s. A system having both *AND* tasks and *OR* tasks is an *AND/OR* precedence-constrained system [3-7].

Many real life applications benefit from the *AND/OR* precedence constraints model, especially when the time limit factor is involved. For example, the goal of an assembly sequencer is to produce a sequence that completely decomposes the original product into its individual parts. If there is a time limit to disassemble the product, it is more likely to disassemble main parts or groups of parts first. Then, if any time is left, disassemble the rest of the product. The first step must be performed following precedence order. The second step is done at any order providing best performance in the available time. This model can be represented in the form of *AND/OR DAG*s, where the direct predecessors of the *OR* tasks indicate which tasks are to be scheduled in the first then in the second steps. To apply the *EED* timing constraint over the *AND/OR* precedence constraint system, a single start and end tasks are introduced by modifying the *AND/OR DAG* that represents the task system model to contain single start and single end tasks where the *EED* is placed between them. An *ICT* is proposed to handle the problem of missing the *EED* in the presence of a transient overload. The proposed *ICT* starts by a feasibility check algorithm to detect the system dependability in the early stages of the scheduling process. The foundation of the scheduling framework described here is a simple but flexible scheduling model. Instead of dealing with specific applications, abstract task systems are used. This leads to a better understanding of the system model and the corresponding scheduling problem. The rest of this paper is organized as follows: Section 2 shows the evolution of task representation. Section 3 gives a general scheduling overview. Section 4 describes different implementations of the imprecise computation technique, and compares the traditional imprecise scheduling technique to the proposed imprecise scheduling technique. Section 5 gathers the proposed work all together. Section 6 contains performance evaluation through simulation and Section 7 draws conclusions and recommends future work.

2. Task representation

The representation of tasks in a real-time system has three major models:

- *Linear chain:* when tasks are independent of each other, a single task may be viewed as a linear chain of subtasks. Each task may execute sequentially on many different processors as each of its subtasks represents a segment of the main task that executes on one of the processors. When sequential precedence constraints exist among tasks, they follow the same rules used with the linear chain of subtasks. This model is simple and helped many researchers to immerge in the study of distributed real-time systems [8] to develop powerful scheduling algorithms and to prove their efficiency through simulation, but it is too narrow to cover all the real-time applications where the precedence constraints among tasks or subtasks are not linear.
- *AND-only graphs*: *AND-only DAG*s were used to permit computation of parallel and distributed processing. An *AND* task is ready to execute when all its direct predecessors are completed. The partial order over these tasks is known as *AND-only* precedence constraints. *AND-only* task representation is widely used in different domains of the real life. Examples are given in [9]: *PERT* (*P*rogram *E*valuation and *R*eview *T*echnique) and *CPM* (*C*ritical *P*ath *M*ethod) models are essential for project scheduling. *AND-only* directed graph model is more general than the linear chain model, but it falls short in describing many real-time applications encountered in practice. In these applications, a task may become ready for execution when some but not necessarily all of its direct predecessors are completed.
- *AND/OR graphs: OR* tasks are added to the previous model to permit more flexibility to the real-time system representation. This model is the general task graph representation because it can be used to represent all possible task models. The concept of *AND/OR* directed graph model is not new, it is used in *Artificial Intelligence* where some problems may be solved in one of many ways [10], it also has some applications in compiler design to represent *SWITCH-CASE* statements [11]. Fault-Tolerant applications may also benefit from *AND/OR* graph scheduling where *OR*

tasks are known as threshold tasks where most error treatment strategies are based on some kind of redundancy or replication [12]. The use of *AND/OR* graphs to represent assembly-planning problems were introduced in [13,14]. There are two types of *AND/OR* scheduling problems: *unskipped* scheduling problem where all predecessors of every *OR* task must eventually be completed, and *skipped* scheduling problem where some *OR* predecessors may be left unscheduled [3].

3. Scheduling

Many optimal scheduling algorithms work in two phases:

- In the pre-processing phase, the given task system is modified to achieve consistency or to remove complicating or conflicting information, such as the precedence constraints.
- In the scheduling phase, the resultant task system is scheduled according to a greedy priority-driven heuristic.

There are, in general, two types of scheduling: scheduling to meet deadlines and scheduling to minimize completion time. Both types of scheduling were investigated in [3] over *AND/OR/Skipped* and *Unskipped* variants through different models of graphs starting from general models and ending by simpler models. It was proven in [3] that: most problems of scheduling tasks with deadlines are *NP-complete* on a single processor, also problems involving the minimization of completion time in a multiprocessor are *NP-complete*, and when *AND* & *OR* tasks are present in the same graph, every tractable problem in the scheduling literature becomes *NP-complete*. All the above justifies the need for heuristic algorithms to solve such problems.

This paper focuses on the pre-processing phase, where the *AND/OR* graph is processed into an *AND-only* graph with special property. A heuristic algorithm –*CljDelete*– is proposed in [3] to solve the problem of scheduling *AND/OR/Skipped* task system to minimize completion time on a single processor. A modified version of *CljDelete*– resulting in *ModCljDelete* heuristic algorithm– is used in this paper to provide us with the schedule tasks with minimum acceptable completion time. This algorithm is used as a starting step for the proposed imprecise computation technique *ICT* shown in the following section. Tasks involved in the resulting schedule are considered to be the mandatory *M* tasks of the system, and tasks left unscheduled are considered to be the optional *O* tasks of the system.

4. Imprecise computation

A real-time system functions properly only in the absence of timing faults. For many such systems, having an approximate but usable result on a timely basis is better than having a late and precise result. An approximate but usable result can often be produced by much less processor time than a precise result. So, imprecise computation technique is applicable whenever a task can produce an imprecise result in less execution time than it would take to produce an exact one in order to conserve resources and eliminate timing faults. This observation is the basis for the imprecise computation technique. By trading off precision for timeliness, the imprecise computation technique prevents missed deadlines by ensuring that an approximate result of an acceptable quality is available whenever the exact result cannot be produced in time. A system based on the imprecise computation technique is called an imprecise system. So, the imprecise computation technique is a way to make a real-time system dependable. By definition [15,16], dependability is the capability of a system to deliver the specified application services during its period of operation. On the other hand it does not forbid the occurrence of failures in general. Dependable real-time scheduling algorithms are capable of achieving high deadline compliance and high predictability. Deadline compliance represents the probability that the system will meet a task's time constraints. Predictability represents the system's ability to decide the feasibility of meeting the time constraints of a given task from a task set well ahead of the deadline. A dependable real-time system produces predictable results even when the timing constraints or the system environment varies.

A task in an imprecise system can be implemented using one of three methods [17]:

- *The Milestone Method*: A task system and its underlying computation algorithm are said to be monotone if the quality of the intermediate result produced by it is non-decreasing as it executes longer. A monotone task system produces a precise result when the entire system completes. An approximate result can be made available by recording the intermediate results produced by the system at approximate instants of its execution, i.e., *milestones*.
- *The Sieve Method*: A computation or a set of computations whose sole purpose is to produce outputs that are at least as precise as the corresponding inputs is called a *sieve* function. If a *sieve* function is executed, it improves the accuracy of its inputs. If it is skipped, processing time is saved but at the expense of having less accurate values. Hence a task that carries out a *sieve* function is optional. It is either completely executed or entirely skipped.
- *The Multiple-Version* Method: This method provides at least two versions of the task system: the primary version and the alternate version(s). The primary version produces a precise result but has longer processing time while an alternate version has a shorter processing time but only produces an approximate and acceptable result. The alternate version corresponds to the mandatory part of the task system. The difference between the two versions defines the optional part. When multiple versions are used, it is necessary to decide before the task starts which version will execute. Also, the optional part will be completely executed (primary version) or not executed at all (alternate version). This requirement is also called a 0/1 constraint.

The *AND/OR* scheduling model can be used to allow real-time systems to function correctly under transient overload, by omitting certain portions of the task system in order to meet hard real-time deadlines. Presumably, under normal operating conditions the full task system is executed and all the deadlines are met. This is done using the *Unskipped* scheduling model where all the tasks must eventually be completed. When an overload occurs (i.e. when the processor utilization exceeds 100%) and the processor can no longer meet all the deadlines, some portions of the task system may be skipped in order to allow critical tasks to meet their deadlines. The *AND/OR/Skipped* task model can represent the portions of the task system that may be skipped. It can represent applications where the precision increases in discrete steps as a task is either executed or skipped as a whole and not partially executed, and this is presumably more common in real-time applications. Our intention is to make a liaison between the scheduling of *AND/OR* task systems and the concept of imprecise computation in order to enhance the performance of the system by decreasing its failure rate. So, we propose an imprecise computation technique *ICT* that can be applied on a real-time system with *AND/OR* precedence constraints. This technique can be applied on general graphs. A global End-to-End deadline *EED* is introduced to this task system between a start node and an end node in the graph. The proposed imprecise computation technique differs from the traditional one in several points such as:

- The traditional model is applied on system composed of independent tasks, each task is formed of a linear chain of subtasks, while the proposed model is applied on systems that can be represented on the form of a general *AND/OR* graph.
- In the traditional imprecise computation technique, a task is decomposed into mandatory subtask and optional subtasks. All the mandatory subtasks must be executed for the system to produce correct results. The execution of optional subtasks enhances the performance of the system as long as the end-to-end deadline is not violated. On the other hand, in the proposed model, a task is dealt with as a unit that cannot be decomposed. Tasks are identified as mandatory *M* or optional *O* tasks according to their type (*AND* or *OR* tasks) as well as their position in the graph. For instance, a maximal task (which is a task with no successors) is always an *M* task, while a minimal task (which is a task with no predecessors) can be an *O* task. Also an *AND* task can be an *O* task when it is a direct predecessor of an *OR* task in the graph, while

the direct predecessors of an *M AND* task are all *M* tasks.

- The end-to-end deadline in the traditional imprecise computation model is defined as a special deadline for each task from the starting subtask to the end subtask. In this case a set of deadlines $D=[d_1,d_2,\dots,d_n]$ is associated with the set of tasks $T=[T_1,T_2,\dots,T_n]$. The global end-to-end deadline *EED* of the proposed model is defined over the whole system between a starting task and an end task of the system.
- The best way to implement a traditional imprecise computation model is using the *milestone* method, as it gives great flexibility to the scheduling algorithm. This method cannot be used in the *AND/OR* precedence constraint model. Rather, the *sieve* model is preferred in this case because it works on the concept of 0/1 constraint that is more convenient as each task is dealt with as a complete unit that cannot be decomposed. It is either completely executed or entirely skipped.

5. The proposed solution

This section introduces the proposed complete solution that includes a new technique to schedule a task system with *AND/OR/Unskipped* precedence constraints in an off-line mode, taking into consideration the existence of a global *EED* over the whole system, to obtain an imprecise result with minimum number of skipped tasks, i.e. with maximum possible quality. This problem-solving concept is not new, it is known in the literature as "*Design-To-Time Real Time Scheduling*" [18]. It is an approach to real-time problem solving in situations where multiple methods exist for many tasks that a system needs to schedule. It involves designing a solution to a problem that uses all available resources to maximize the solution quality within the available time.

5.1. System model

This section introduces the basic terminology needed for the system model used in this paper. It represents a real-time system consisting of a general task graph and applying the global *EED* concept.

- The system consists of a set of *m* identical processors $P = \{P_1, P_2,\dots,P_i,\dots,P_m\}$ and a set of *n* tasks $T = \{T_1, T_2,\dots,T_i,\dots,T_n\}$.
- Each task T_i executes on one processor for pt_i units of time. So a set of processing times $pt = \{p_1,p_2,\dots,p_i,\dots,p_n\}$ is associated with the set of tasks *T*.
- There is a partial order < defined over *T*.

- If $T_i < T_j$, then T_i is a *predecessor* of T_j, and T_j is a *successor* of T_i.
- T_i is a *direct predecessor* of T_j and T_j is a *direct successor* of T_i if there is no T_k such that $T_i < T_k < T_j$.

- The partial order < is said to be *in-forest* if whenever $T_k < T_i$ and $T_k < T_j$, either $T_i < T_j$ or $T_j < T_i$, and it is said to be *in-tree* if it has a unique element with no successors.
- T_i is an *AND* task if its execution may begin only after all its direct predecessors are completed, and T_j is an *OR* task if its execution may begin after any one of its direct predecessors is completed.
- The partial order is also represented by a weighted an transitively reduced directed graph $G = (T, E, pt, \Pi)$, called the *task graph*.

- In this graph, there is a vertex T_i for every task in the set *T*.
- The set *E* is known as the *set of edges*: if T_i is a direct predecessor of T_j in the partial order, then (T_i , T_j) is an element of *E*.
- The set $pt = \{pt_1, pt_2,\dots,pt_i,\dots,pt_n\}$ denotes the *set of processing times*.
- The set $\pi = \{\pi_1, \pi_2,\dots, \pi_i,\dots, \pi_n\}$ denotes the set of thresholds. It indicates the number of direct predecessor tasks that must be completed before a task may begin execution.

- A task graph together with a global end-to-end deadline *EED* (between a start task and an end task) is denoted (G,EED). This 2-tuple characterizes a scheduling problem; it is called a *task system*.
- A task with no successors is a *maximal* task, and a task with no predecessors is a *minimal* task.

- All the maximal tasks in a task graph are classified as *essential* tasks; this means that they must be executed.
- If an *AND* task is *essential*, then its direct predecessors are *essential*.

- If an *OR* task T_j is *essential*, then the scheduling algorithm must choose one direct predecessor T_i to be *essential* and the precedence constraint $T_i < T_j$ must be obeyed in scheduling the task system.
- If a task is not classified as *essential*, it is *inessential*.

a- In a *skipped* task system, *inessential* tasks may be skipped, that is, they do not have to be executed.

b- In an *unskipped* task system, *inessential* tasks must eventually be completed.

5.2. ModCljDelete heuristic algorithm

Modifications on the *CljDelete* algorithm proposed in [3] resulting in *ModCljDelete* algorithm are:

- First: the set of tasks *T* can be represented as a vector T_{nx1} (of length *n* which is the number of tasks in the graph) where a 2 means an *OR* task and a 1 means an *AND* task. The index of T_{nx1} itself will be the *id* of the task. In this way, there is no need for the set of task types *kind[n]* in the input.
- Second: to represent the set of edges *E*, we can benefit from the Adjacency Matrix *AM* concept, to eliminate the multiple use of the depth-first-search *DFS*, as it will be direct retrieval from the *AM*. $AM[i,j]=1$ when an arc exists from T_i to T_j and 0 otherwise. Some of the advantages of this method are: the ease of implementation, as there is no need for a large number of queues and id's, as well as many repetitions of the *DFS*, the time complexity to retrieve data from the AM_{nxn} or $Path_{nxn}$ matrices is usually *O(1) and* at most never exceeds $O(n^2)$, while the time complexity of the *CljDelete* algorithm was $O(n^2(m+n))$ [3], where *n* is the number of tasks and *m* is the number of arcs in the task graph.

This algorithm results in the set of mandatory tasks of the system. It is also used as a feasibility check algorithm.

The *ModCljDelete* algorithm's pseudo code is as follows:

```
Input: Vector of tasks $T_{nx1}$ and $AM_{nxn}$ to represent the AND/OR graph, set of processing times $pt_{nx1}$.
Output: Vector of tasks $T_{nx1}$ and $AM_{nxn}$ to represent the Mandatory tasks.
Variables: integer $length_{nx1}$, $ordegree_{nx1}$, $cost_{nx1}$, $sum_{nx1}$, $Path_{nxn}$, OR_Queue,.
Start:
p. Copy $AM_{nxn}$ into $Path_{nxn}$.
   Form $Path_{nxn}$ using the following transitive closure algorithm:
   For (k=1; k≤n; k++)
       For (i=1; i≤n; i++)
           For (j=1; j≤n; j++)
               Path[i,j]= Path[i,j] OR (Path[i,k] AND Path[k,j])
0. Copy the processing time $pt_{nx1}$ into $length_{nx1}$.
   For each OR task k, where T[k]=2:
     a. check the k's column of Path (Path[i,k], for all i=1 to n), tasks i's
        with Path[i,k]≠0 are all the predecessors of the OR task k.
     b. if all predecessors i's are AND tasks (T[i]=0), or no predecessors exist for
        task k, then put k in the OR_Queue.
1. If OR_queue is empty, quit(done).
2. Initialize to zero all ordegree[ ] counters associated with tasks.
3. For each task k in the OR_Queue
     The non-zero element in A[j,k], for all j=1to n, represents an AND predecessor of task k.
     The ordegree[j] equals to the sum of all the direct successors of task j, i.e.:
         $ordegree[j]=\Sigma_{\text{for all } i=1 \text{ to } n} AM[j,i]$.
4. Compute floating-point cost ratio for each AND task i.
         cost[i] = length[i]/ordegree[i]
5. Find direct predecessors j's of each task in the OR_Queue k from AM[j,k].
6. Find predecessors i's of tasks j's from Path[i,j], for each one perform:
         sum[j] = sum[j] + cost[i].
7. Find task i where sum[i] is the largest number in the $sum_{nx1}$.
8. From A[i,k] find the direct OR successor k of task i (where T[i]=2)
     a. let A[i,k]=0 (i.e. remove the arc).
     b. If task k has now only one predecessor
        (only one non-zero element in A[j,k] for
        all j=1 to n) then remove task k from
        the OR_Queue and let T[k]=0.
9. If the OR_Queue is empty then refill the OR_Queue with a new set of minimal OR tasks.
10. Go to step 1.
```

5.3. Feasibility check algorithm

The set of tasks is considered to be feasible whenever the completion time of its subset of mandatory tasks doesn't exceed the given *EED*. Otherwise the set of tasks is infeasible. At the end of *ModCljDelete* algorithm, the total execution time of mandatory tasks *minproc* is computed and compared with the *EED*.

- The task system is infeasible if *minproc* > *EED*, in this case a FAILURE occurs with no possible recovery with our solution unless the user changes some of the system configurations.

- Otherwise, it is feasible if *minproc* ≥ *EED*, and SUCCESS occurs with different degrees of quality.

5.4. Implementation of ICT

After the *AND/OR* graph is generated by *RGG* or given by the user, the *ModCljDelete* is applied over the graph. At the end of this algorithm, the total execution time of the *M* tasks *minproc* is computed and compared with the *EED*.

- The system is infeasible if *minproc* >*EED*, in this case a FAILURE occurs. No recovery is reached unless the system's configuration is changed.
- Otherwise, it is feasible if *minproc* ≤ *EED*, and SUCCESS occurs with different degrees of quality. In this case, an *ICT* is followed:

- If *minproc* = *EED*, then no more improvement can be done. An imprecise result with maximum error is found.
- If *minproc* < *EED*, then a recursive algorithm is followed to enhance the system performance and the result quality. *O* tasks to be removed from the schedule are taken from the bottom of the list sorted using one of five methods.

The *ICT* algorithm's pseudo code is as follows:

```
Input:   from RGG or by user: $n$, $T_n$, $AM_{nxn}$ $pt_n$, EED,
from ModCljDelete: Man, Totalproc, minproc
Output: Scheduled list of tasks $T_n$ , Fraction of Discarded
Work Frac1, Fraction of Skipped Tasks Frac2.
Variables:  Tot = copy of Totalproc, Copyn = copy of
number of tasks n.
Start:
0.  Compare Totalproc with EED:
        if Totalproc ≤ EED
          then → SUCCESS with No Error →STOP.
1.  else Compare minproc with EED:
        if minproc > EED
          then → FAILURE with No Recovery → STOP
    else
    if minproc = EED
          then → SUCCESS with Maximum Error
               Frac1 = (Totalproc - minproc) /
          Totalproc
               Frac2 = (n – Man ) / n          → STOP
2.  else ITC is followed:
    Sort Optional tasks according to one of five priority
    protocols.
    a.  Priority Protocol (1): Sort O tasks
        ascendingly according to their level in
        the graph.
    b.  Priority Protocol (2): Sort O tasks descendingly
        according to their level in the graph.
    c.  Priority Protocol (3): Sort O tasks descendingly
        according to their processing times.
    d.  Priority Protocol (4): Sort O tasks ascendingly
        according to their processing times.
    e.  Priority Protocol (5): Sort O tasks descendincly
        according to their priorities given by the user.
3.  Tot = Totalproc
    Copyn = n
4.  Remove an optional task T[copyn] from the bottom
    of the sorted list.
5.  Tot = Tot – tp[T[copyn]]
6.  if EED < Tot
      then Copyn = Copyn – 1
           goto step 4.
7.  else → SUCCESS with Imprecise Solution
           Frac1 = ( Totalproc – Tot ) / Totalproc
          Frac2 = ( n – Copyn) / n
8.  Output final Schedule = Mandatory tasks + used
Optional tasks.
    Output error = Frac1 , Frac2.
```

Performance is measured using [8]:

- The fraction of discarded work *Frac1* = [Σ *pt*s of skipped *O*s]/[Σ *pt*s of all tasks], or
- The fraction of skipped tasks *Frac2* = [number of skipped *O*s]/*n*.

5.5. Random Graph Generator (RGG) algorithm

An algorithm *RGG* to generate random graphs with *AND/OR* precedence constraints to be used as inputs for the proposed algorithms is introduced in this section. *RGG* is based on a rich set of parameters resulting from the combination of the lists of simulation parameters given in [7,16]. Our method is designed having in mind that most of the parameters have to be chosen at random as much as possible to provide a wide range of non-repeated graphs. Important parameters are:

- *q*: the ratio of actual number of edges to the total possible number of edges in the graph.
- *p*: the ratio of actual number of *OR* tasks to the number of candidate *OR* tasks in the graph.
- *GridRatio*: the ratio representing the X/Y grid controlling the graph.
- *[L,U]*: a uniform distribution for the lower and upper levels of the processing time values.

A pseudo code of the ***RGG*** algorithm is shown as follows:

```
Input: float q, p, GridRatio, L, U.
Output: int n, AM_nxn, T_nx1, float pt_nx1
Variables: int ActualEdges, Edges, X, Y, X_nx1, Y_nx1,
CandOR, nOR, n1, n2.
Start:
1.  Seed the random number generator.
2.  Pick up number of tasks n at random in the
    range [1,200], and be sure n>zero.
3.  Calculate total number of possible edges in the
    system graph:
        a. n1 = number of nodes in each
           level Y[i]
           n2 = number of nodes in all of the following
        levels starting from level Y[i+1]
           n1 x n2 = all possible edges coming out of
    the nodes in level Y[i]
        b. Edges = ∑results of all the
           previous multiplications
4.  Calculate actual number of edges in our graph:
          ActualEdges = q x Edges
5.  Pick up a random number X in the range [1,50] to
    represent number of columns in the X by Y grid.
    Again make sure that X>zero.
6.  Calculate Y = X/GridRatio.
7.  For each task i from 1 to n, withdraw a random
    number X[i], and another Y[i] to represent where i is
    placed in the grid. A check is made so that X[i]>zero
    and Y[i]>zero. Another check is made to prevent
    repetition of the same couple (X[i],Y[j]) with different
    tasks.
8.  Tasks are sorted in an ascending order according to
    the values of Y[j], and then are sorted in an
    ascending order according to the values of X[i]
    within the same value of Y[j].
9.  The AM_nxn is initialized by filling it with zeros.
10. Edges to be connected in AM_nxn are chosen at
    random providing their number doesn't exceed the
    previously calculated value of Edges.
11. Task processing times pt_nx1 are chosen at random in
    the interval [L,U].
12. Initialize T_nx1 =1 as all tasks are still AND tasks.
13. Identify candidate-OR tasks to be those which have
    two or more direct predecessor tasks. Number of
    those tasks = CandOR.
14. Number of actual OR tasks  nOR= p x CandOR
15. Generate OR tasks at random from the
    candidate OR tasks, providing that their sum
    doesn't exceed nOR.
16. Update the entries in T_nx1 of the OR tasks to be 2.
17. If there are more than one minimal task in the
    generated graph:
        a. Add a dummy AND task T0 with pt[0]=0
           and T[0]=1.
        b. Make this task the direct predecessor of all
           minimal tasks.
18. If there are more than one maximal task in the
    generated graph:
        a. Add a dummy AND task T_n+1
           with pt[n+1]=0 and T[n+1]=1.
        b. Make this task the direct
           successor of all maximal tasks.
```

6. Simulation

The simulation consists of four different experiments. Each experiment is run on different sets of parameters. The first experiment is concerned with the verification of *ModCljDelete*. The three remaining experiments are used for the performance evaluation of the proposed *ICT*. In each experiment, a different measure of performance is used: the *failure rate -FR-* in the second experiment, the *fraction of discarded work -Frac1-* in the third, and the *fraction of skipped tasks -Frac2-* in the fourth. The simulation model is validated relative to those measures of performance that can be used in decision-making.

6.1. Experiment (1): modcljdelete verification

In order to verify this algorithm, it is compared with the original *CljDelete* on the basis of a performance ratio defined in [3]. Fig. 1 shows that both *CljDelete* and *ModCljDelete* have very close results. This experiment indicates that *ModCljDelete* is verified and can be used as a tool to find the mandatory tasks of the system in our proposed *ICT*.

6.2. Validation of proposed ICT

Simulation parameters –explained in section 5.5- are: *q*, *p*, *X/Y* and *[L,U]*. Four different sets of parameters, shown in table 1, are used with 3 experiments with different measures of performance. Number of tasks *n* varies from 10 to 200. The values of *EED* are chosen depending on those of *n* and *[L,U]*.

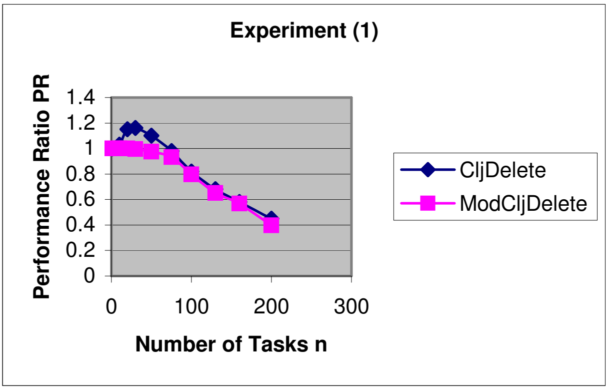

Fig.1. ModCljDelete vs CljDelete.

Table 1
Simulation sets of parameters

Parameters	Set(1)	Set(2)	Set(3)	Set(4)
Q	0.2	0.5	0.5	0.5
P	0.1	0.5	0.8	1.0
X/Y	0.1	0.5	0.5	0.5
[L,U]	[1,10]	[1,100]	[1,100]	[1,100]

6.2.1. *Experiment (2): failure rate*

For each *EED* value at each number of tasks *n*, 100 runs are done at random and the number of failure times are counted and denoted by *NF*. The failure rate *FR* is computed as follows: *FR* = *NF*/100. This experiment sets a comparison between the normal and the imprecise frameworks. In all cases, the imprecise outperforms the normal framework. Figs. 2 and 3 gives samples of the outputs to this experiment. The complete output is found in [19]. Parameters' Set (1) gives an almost *AND-only* graph. When there are few tasks in the system (10 or 20), the imprecise framework gives better results than the normal framework. When the number of tasks goes larger (100 and 200), the difference in performance between the two frameworks is barely noticed. In the worst cases though, the normal framework never outperforms the imprecise framework. Set (2) provides a moderate number of *OR* tasks in the graph, and in Set (3), this number is much higher. As the number of *OR* tasks increases, there is a larger set of optional tasks which gives more flexibility to the imprecise framework. Set (4) shows the case of all candidate *OR* tasks becoming actual *OR* tasks. It is to be noticed that as long as *p* increases, the corresponding value of *EED* in the imprecise framework, which gives a good probability of acceptable solution decreases.

6.2.2. *Experiment (3): fraction of discarded work frac1*

Frac1 is the ratio between the sum of execution times of skipped tasks and that of all the tasks in the system. In this and the following experiment, 5 different methods – shown in table 2– are used to sort *O* tasks so that the worst task in each case is removed first. Sample outputs are shown in figs. 4 and 5, and the complete output is in [19].

In Set (1), where *p* is very low, sudden changes occur in the value of *Frac1* from 1 to 0. This is due to the small number of optional tasks, so there is a lack of flexibility in the system.

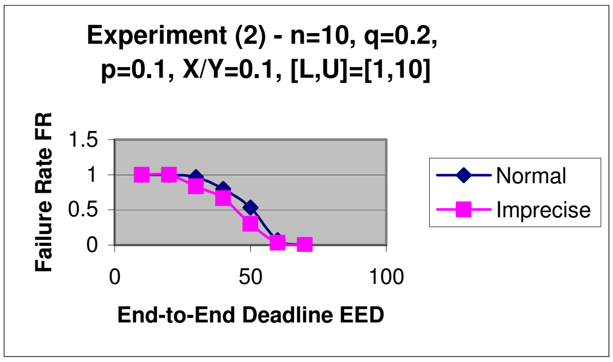

Fig. 2. Experiment (2), Set (1), *n*=10.

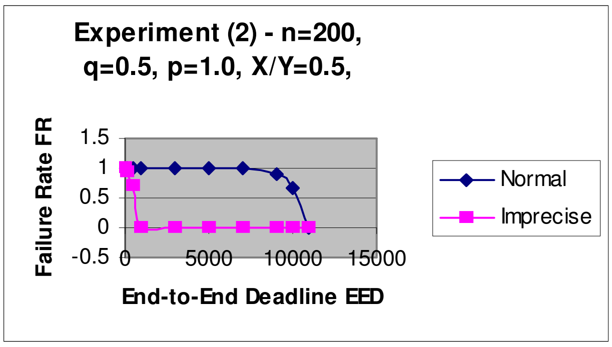

Fig. 3. Experiment (2), Set (4), *n*=200.

Table 2
Different O tasks sorting methods

METHOD	Sorting O tasks	Worst task
Method1	↑ w.r.t graph level	Highest level
Method2	↓ w.r.t. graph level	Lowest level
Method3	↓ w.r.t. processing times	With lowest pt
Method4	↑ w.r.t. processing times	With highest pt
Method5	↓ w.r.t. given priority	Lowest priority

In Sets (2)→(4), *p* is getting higher which provides smoother curves due to the increase in the number of optional tasks. The system becomes more flexible and the value of *Frac1* decreases gradually with the increase in *EED*.

Best performance is obtained when all candidate *OR* tasks are becoming actual *OR* tasks.

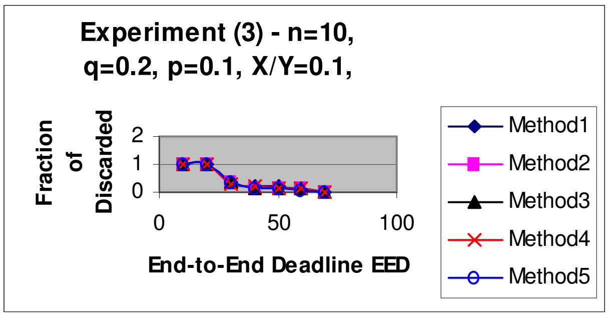

Fig. 4. Experiment (3), Set (1), *n*=10.

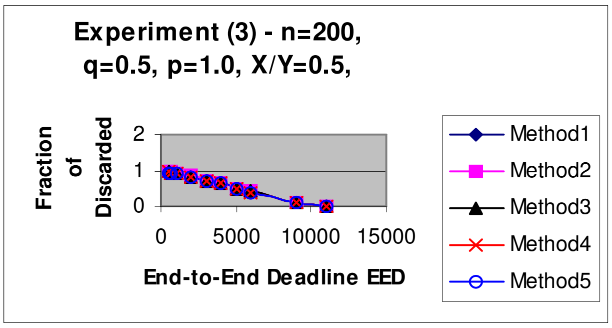

Fig. 5. Experiment (3), Set (4), *n*=200.

6. 2. 3. Experiment (4): fraction of skipped tasks frac2

Frac2 is the ratio between the number of skipped tasks and that of all the tasks in the system. Sample outputs are shown in figs. 6 and 7, also the complete output is in [19]. In this experiment, it is to be noticed that Method4 outperforms all other methods with all sets of parameters and all number of tasks, especially when *p* gets higher. This is special for this specific measure of performance: it is always true when the task with the highest processing time is removed first, then most probably the largest number of tasks scheduled in the system among all other methods will be obtained.

7. Conclusions and future work

This paper devises an algorithm to schedule tasks with *AND/OR/Unskipped* precedence constraints and introduces a method to integrate precedence constraints

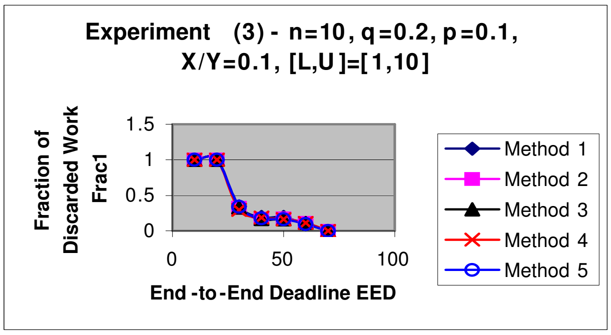

Fig. 6. Experiment (4), Set (1), *n*=10.

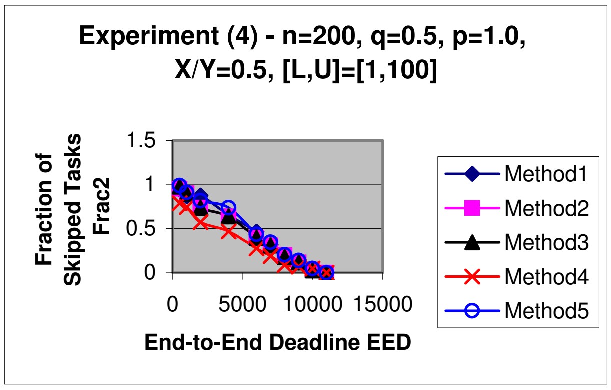

Fig. 7. Experiment (4), Set (4), *n*=200.

and timing constraints into the same scheduling problem. An imprecise computation technique *ICT* is proposed to handle the problem of missing the *EED* in the presence of a transient overload. *ICT* provides system *dependability*. It tends to achieve *deadline compliance* as much as possible, it gives lower failure rates compared with the corresponding normal scheduling model. Its feasibility check helps *predicting* whether or not a certain system can meet its timing constraints at the very early stages of the algorithm.

There are several promising areas for further work, for example: application in the on-line mode, use of multi-processors scheduling to enhance the system performance, and subtasks scheduling of each individual task within the proposed framework.

References

[1] M. A. Austin, "Parallel Distributed Real-Time Systems in Manufacturing (An Aerospace View)", Pratt and Whitney,

United Technologies, Proceedings of the 1997 Joint Workshop on Parallel and Distributed Real-Time Systems, Institute of Elec-trical and Electronic Engineers (1997).

[2] R. Rajkumar, "Synchronization in Real-Time Systems: A Priority Inheritance Approach", Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Kluwer Academic Publishers, Boston Hardbound, August (1991).

[3] D. W. Gillies and J. W. S. Lui, "Scheduling Tasks with AND/OR Precedence Constraints", Second Annual IEEE Symposium on Parallel Distributed Processing, December, pp. 379-387 (1990).

[4] D. W. Gillies and J. W. S. Lui, "Scheduling Tasks with AND/OR Precedence Constraints", Report No. UIUCDCS-R-90-1627 (UIUC-ENG-176), Department of Computer Science, University of Illinois at Urbana-Champaign (1991).

[5] D. W. Gillies, "Algorithms To Schedule Tasks With AND/OR Precedence Constraints", Ph. D. Thesis, University of Illinois, Urbana (1993).

[6] D. W. Gillies and J. W. S. Lui, "Scheduling Tasks with AND/OR Precedence Constraints", SIAM Journal on Computing (1993).

[7] D. W. Gillies and J. W. S. Lui, "Scheduling Tasks with AND/OR Precedence Constraints", Office of Naval Research (1995).

[8] J. Sun, "Fixed-Priority End-To-End Scheduling In Distributed Real-Time Systems", Ph. D. Thesis, University of Illinois, Urbana-Champaign (1997).

[9] L. Badvicka and M. Berka, "A Tutorial On Networks: PERT & CPM", Faculty of Civil Engineering, Eunet Czchia, Czech Republic (1997).

[10] N. J. Nilsson, "Principles Of Artificial Intelligence", Palo Alto, California, Tioga Publishing Company (1980).

[11] S. A. Warshall, "A Theorem On Boolean Matrices", Journal Of The ACM, Vol. 9, pp. 11-12 (1962).

[12] P. Chevochot and I. Puaut, "Scheduling Fault-Tolerant Distributed Hard real-Time Tasks Independently of The Replication Strategies", IRISA, Campus de Beaulieu -35042 Rennes- France (1999).

[13] L. S. H. DeMello and A. C. Sanderson, "AND/OR Graph Representation Of Assembly Plans", Proceedings of AAAI, Philadelphia, PA, Aug 11-13 Vol. 2, pp. 1113-1119 (1986).

[14] J. D. Wolter, "On The Automatic Generation Of Plans For Mechanical Assembly", Ph. D. Thesis, University of Michigan, September (1998).

[15] J. Peleska, "On The Unified Formal Approach For The Development of Fault Tolerant and Secure Systems", Nordic Seminar on Dependable Computing Systems, Technical University of Denmark (1996).

[16] S. Shekhar and B. Hamidzadeh, "SARTS: A Dependable Real-Time Search Algorithm", Department of Computer Science, University of Minnesota, Minneapolis, IEEE Transactions on Knowledge and Data Engineering (1995).

[17] D. L. Hull, W. Feng and J. W. S. Liu, "Enhancing The Performance and Dependability of Real-Time Systems", IEEE International Computer Performance and Dependability Symposium, Erlangen, Germany, April (1995).

[18] A. Garvey and V. Lesser, "Design-To-Time Real-Time Scheduling", IEEE transactions on Systems, Man and Cybernetics - Special Issue on Planning, Scheduling and Control, Vol. 23 (6) (1993).

[19] Hanan H. El Meligy, "Imprecise Computation Technique To Schedule AND/OR tasks with Global End-to-End Deadline In Distributed Real-Time Systems", Master Thesis, Alexandria University, Faculty of Engineering, Computer Science and Automatic Control Department (2002).

Received June 19, 2002
Accepted October 14, 2002