Acta Cybernetica **18** (2007) 239–255.

Synthesising Robust Schedules for Minimum Disruption Repair Using Linear Programming

Dávid Hanák[*], Nagarajan Kandasamy[†]

Abstract

An off-line scheduling algorithm considers resource, precedence, and synchronisation requirements of a task graph, and generates a schedule guaranteeing its timing requirements. This schedule must, however, be executed in a dynamic and unpredictable operating environment where resources may fail and tasks may execute longer than expected. To accommodate such execution uncertainties, this paper addresses the synthesis of robust task schedules using a slack-based approach and proposes a solution using integer linear programming (ILP). Earlier we formulated a time slot based ILP model whose solutions maximise the temporal flexibility of the overall task schedule. In this paper, we propose an improved, interval based model, compare it to the former, and evaluate both on a set of random scenarios using two public domain ILP solvers and a proprietary SAT/ILP mixed solver.

Keywords: scheduling, integer linear programming, robustness, slack, resource failure

1 Introduction

Scheduling plays a crucial role in manufacturing and service industries where companies must sequence their activities (or tasks) appropriately to meet customer deadlines. An off-line scheduling strategy considers resource, precedence, and synchronisation requirements of tasks, and generates a static schedule satisfying task timing constraints [2]. This schedule executes in a dynamic and unpredictable operating environment where critical resources may fail, tasks may execute longer than expected, or certain new tasks may need urgent processing. Consequently, the task schedule must accommodate such execution uncertainties.

[*]Department of Computer Science and Information Theory, Budapest University of Technology and Economics (BME SZIT), Hungary, E-mail: dhanak@cs.bme.hu

[*]Computer and Automation Research Institute of the Hungarian Academy of Sciences (MTA SZTAKI), E-mail: dhanak@sztaki.hu

[†]Department of Electrical and Computer Engineering, Drexel University, Philadelphia, PA, U.S.A., E-mail: kandasamy@cbis.ece.drexel.edu

In this paper we address the synthesis of robust task schedules using a slack-based approach. In [11] we developed a method to construct schedules where individual tasks retain some temporal flexibility in the form of slack while satisfying their timing requirements. As opposed to *reactive* methods [7,16,19], which recover from the disruption as it happens, our method was *proactive,* i.e. it constructed a schedule that could absorb some disruptions without the need for rescheduling. In previously proposed proactive methods, like [4] and [9], a given amount of slack had been added to the tasks to accommodate expected repair times prior to scheduling. This had resulted in an increased make-span of the entire schedule. On the other hand, our method assumed that the tasks had explicit deadline and resource requirements. The goal was then to maximise the slack of each task such that the resulting schedule satisfied all the temporal constraints and the flexibility was maximal according to a given cost function. We proposed an Integer Linear Programming (ILP) model of the scheduling problem, and evaluated it using two different ILP solvers. In this paper, we advance our method by introducing an improved ILP model that performs better on larger problems. We also get a third kind of ILP solver involved in the evaluation of the new model, and compare the evaluation results with those of the first model.

We begin the discussion in Sect. 2 with a brief introduction of the background and preliminary assumptions. Section 3 summarises the first, slot based ILP model formulated in [11] and introduces the second, interval based model. We also show how the new model can incorporate additional forms of temporal constraints. Section 4 evaluates the performance of both models and compares the evaluation metrics. We conclude the paper with a discussion of future work in Sect. 5.

2 Preliminaries

This section introduces the application domain, then goes on to discuss the task model, sources of slack in a task schedule, and the slack distribution algorithm.

2.1 Application Domain

A research group of the *Institute for Software Integrated Systems (ISIS)* at Vanderbilt University in Nashville had been participating in the *Autonomous Negotiating Teams (ANT)* project [1, 17] of the Defense Advanced Research Projects Agency (DARPA) for several years, in cooperation with the *Information Sciences Institute (ISI)* at the University of Southern California in Los Angeles. The deliverable result of the research was the prototype of a software tool to aid the scheduling of flight missions and regular maintenance of Harrier aircraft in a U.S. Marine Corps squadron.

A considerable part of the effort was the development of the core scheduling engine for maintenance tasks, which has been solved by implementing a finite domain constraint scheduler in Mozart-Oz [18]. This approach relies on a group of *propagators* which act independently on a shared set of variables with finite in-

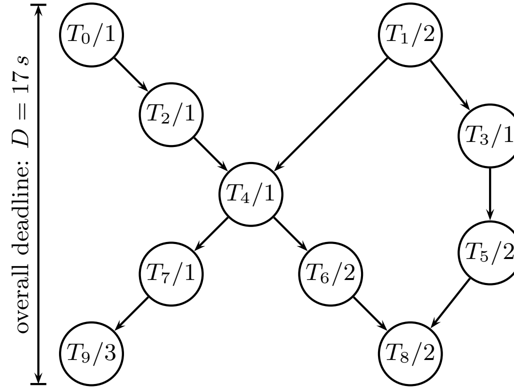


Figure 1: An example task graph G with an overall deadline of 17 seconds

teger domains [20], which, however, due to the independence of the propagators, can sometimes lead to inconveniently deep but solutionless subtrees in the search tree of the entire problem. This in turn can produce an unpredictable behaviour, where the scheduling engine produces a solution in a matter of seconds for one problem, but enters into a unacceptably long and fruitless search for another. This problem is a current issue in the field of constraint programming and it has been addressed by various research efforts [3, 15]. Our response to the problem was the attempt introduced in this paper to replace the domain of finite domain constraint programming with *integer linear programming,* which is perhaps more reliable in this respect. A further advantage of the approach is that it gives us an *anytime algorithm* [5] "for free".

2.2 Modelling Assumptions

Figure 1 shows a directed acyclic graph G modelling task interaction. Tasks are non-preemptive and have resource, precedence, and synchronisation requirements. The graph comprises vertices and edges representing tasks and precedence constraints, respectively. Each vertex is labelled T_i/c_i, where T_i is a task and c_i its estimated execution time in appropriate time units (seconds in this example). We denote the precedence constraint between tasks T_i and T_j in the graph by $T_i \rightarrow T_j$. Tasks without predecessors are called entry tasks and tasks having no successors are called exit tasks. We also assume that each task T_i requires a set of resources $\mathcal{R}_i = \{R_m\}$ for its execution where R_m denotes a resource of type m. Also, for each resource R_m, its available capacity at a given t point in time is given by $\text{cap}(R_m, t)$.
Scheduling is a mapping of tasks to resources such that the specified precedence and deadline constraints are satisfied. The desired result is a feasible schedule specifying the *start times* (also referred to as *release times*) for each task T_i. It is also necessary to introduce some slack in this schedule to improve its robustness to execution uncertainties. In many cases, the necessary slack may be obtained by

appropriately dividing up the entire available *time frame* (i.e. the interval between the overall start and deadline time, $0\,s$ resp. $17\,s$ in the example) among the tasks.

Assume that tasks T_0 and T_1 start at $0\,s$, and that G must meet a deadline of 17 seconds, i.e. T_8 and T_9 must finish before $17\,s$. Note, however, that the longest path $T_0T_2T_4T_6T_8$ through G is only $7\,s$ long. This implies that a slack of $17 - 7 = 10\,s$ can be distributed to tasks along that path to retain some temporal flexibility during their scheduling. We now discuss a method aimed at distributing G's overall slack among tasks such that the slack added to each intermediate task is maximised. This process results in a *scheduling range* $[\,r_i, d_i)$ for each T_i where r_i and d_i denote the earliest release time and latest deadline, respectively.

2.3 Slack Distribution

Initially, only entry and exit tasks having no predecessors and successors, respectively, have their release times and deadlines fixed. In the slack distribution problem, the overall graph time frame must be distributed over each intermediate task such that all tasks can be feasibly scheduled on their respective resources. Slack distribution is NP-complete and various heuristics have been proposed to solve it. We use the approach proposed in [6] to maximise the slack added to each task in graph G while still satisfying its deadline D. The heuristic is simple, and for general task graphs, its performance compares favourably with other heuristics [12].

As part of the slack distribution, entry and exit tasks in the graph are first assigned release times and deadlines respectively. A path $path_q$ through G comprises one or more tasks $\{T_i\}$; the slack available for distribution to these tasks is:

$$slack_q = D_q - \sum_{i\,:\,T_i \in path_q} c_i \tag{1}$$

where D_q is the length of the time frame of $path_q$ (i.e. the difference between the deadline and the release time of the path) and c_i is the execution time of task T_i along this path. The distribution heuristic in [6] maximises the minimum slack added to each T_i along $path_q$ by dividing $slack_q$ equally among tasks. During each iteration through G, a *non-extensible* path $path_q$ is chosen such that $slack_q/n_q$ is minimal, where n_q denotes the number of tasks along $path_q$. Then the corresponding slack is added to each task along that path. The deadlines (release times) of the predecessors (successors) of tasks belonging to $path_q$ are updated. Tasks along $path_q$ are then removed from the original graph, and the above process is repeated until all tasks are assigned release times and deadlines.

The graph in Fig. 1 is used to illustrate the above procedure. First, we select the path $T_0T_2T_4T_6T_8$ shown in boldface in Fig. 2(a); the total execution time of tasks along this path is $7\,s$, and as per the heuristic, a slack of $(17 - 7)/5 = 2\,s$ is distributed to each task. Once their release times and deadlines are fixed, these tasks are removed from the graph. Then path $T_1T_3T_5$ and finally path T_7T_9 is chosen, as shown in Figs. 2(b) and 2(c), respectively. In the former case, a slack of $\lfloor(13 - 5)/3\rfloor = 2\,s$ is added to each task. Our algorithm leaves any remaining slack ($2\,s$ in Fig. 2(b)) unexploited, although it could be distributed to tasks with longer execution times to allocate the *relative* slack more evenly.

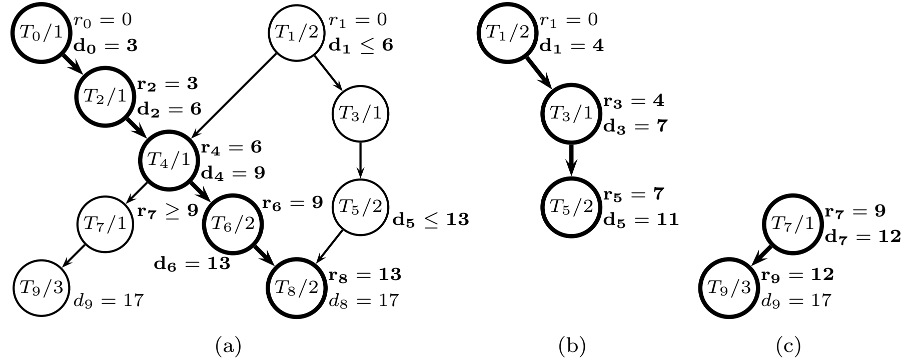


Figure 2: Steps corresponding to the the deadline assignment algorithm in [6]; the selected paths are shown as bold edges

3 ILP Model Formulation

In this section we first describe the part of the scheduling problem that remains after the slack distribution finishes, and present how it can be formulated as an ILP model. Two different formulations are shown, one where a contiguous sequence of uniform length *time slots* is assigned to each task, and another where one of a set of possible *predetermined intervals* is chosen for each task.

3.1 Interval Allocation

Once tasks are assigned deadlines, each T_i has a scheduling range given by $[\,r_i, d_i)$. However, to generate a feasible mapping of tasks to a limited number of resources, these scheduling ranges must be reduced appropriately to account for resource contention during task execution; we adapt concepts from interval scheduling [8] to solve this problem.

The scheduling range for T_i is first decomposed into a number of possibly overlapping intervals $\{I_{ij}\}$. Each I_{ij}, corresponding to the jth possible scheduling interval for T_i is such that:

$$I_{ij} = [\,r_{ij}, d_{ij}) \text{ where } r_i \leq r_{ij} \leq d_i - c_i \text{ and } r_i + c_i \leq d_{ij} \leq d_i. \tag{2}$$

I_{ij} is also assigned a weight

$$w_{ij} = \frac{d_{ij} - r_{ij} - c_i}{d_{ij} - r_{ij}} \tag{3}$$

denoting the scheduling flexibility within that interval in terms of available relative slack.

Robust schedule generation can now be formulated as an *interval selection* problem where exactly one scheduling interval for each task must be selected such

that (i) at any point in the schedule, the overlapping task intervals do not consume more than the number of available resources and (ii) the sum of the interval weights is maximised.

3.2 Slot Based Approach

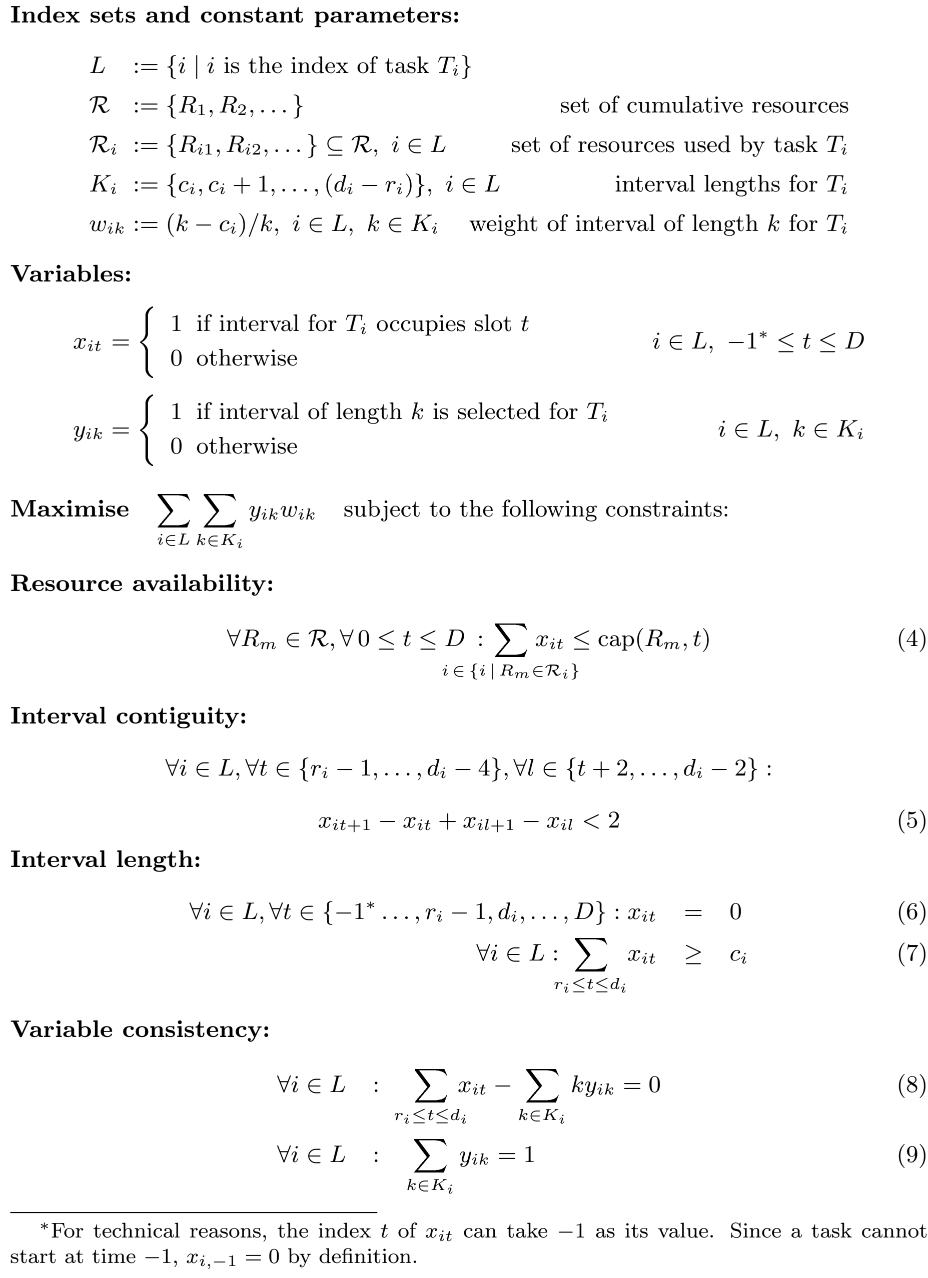
In [11], we proposed an ILP model using uniform length *time slots,* where the solution of a problem is an assignment of a contiguous set of slots to each task, such that the task can be executed in those slots without violating any of the constraints. The interval selection described above appears in this model only indirectly, since an interval is implicitly determined by the set of contiguous slots selected for the task. The model is shown on Fig. 3, and can be explained as follows. A boolean variable x_{it} corresponds to each task-slot pair, such that a specific task is scheduled to run in a particular slot if and only if the corresponding variable is assigned the value 1 in the solution. It is also necessary to introduce a set of auxiliary variables y_{ik}, such that y_{ik} is 1 if and only if the total number of slots (viz. the length of the scheduling interval) assigned to task T_i is exactly k. These values are used in the objective function to mask out the predetermined interval length weights.[1] The constraints ensure that resource capacities are not exceeded (4), that the intervals selected by the slot variables are contiguous to ensure non-preemptive execution (5), that tasks are not executed outside their scheduling ranges (6), and that the generated interval lengths are long enough to accommodate the tasks (7). Equations (8) and (9) describe the connection between the appropriate x_{it} and y_{ik} variables.

The major weakness of this approach is hidden in (5), the constraint which ensures that the scheduling intervals assigned to the tasks are contiguous. This involves moving a simple convolution window over the entire scheduling range and ensuring that there is not more than one 0-to-1 transition. Unfortunately, this is a nonlinear requirement, thus the number of ILP inequalities it can be expressed with is quadratic in the average size of the scheduling ranges, totalling approximately $\sum_i (d_i - r_i)^2$. Another difficult and inefficient detail of the model is the cost function, which cannot be expressed directly in terms of the slot variables, but requires the introduction of a large number of auxiliary variables, precisely $\sum_i (d_i - r_i - c_i)$ many.

3.3 Interval Selectors

To circumvent the shortcomings of the slot based model, the number of equations was reduced at the cost of increasing the number of variables, in the hope that the ILP solvers can cope better with the latter than the former. In the new model, a boolean variable was assigned to each selectable scheduling interval of each task defined in (2). A solution is an assignment of 0/1 values to these variables, such

[1] Weights must be calculated in advance and "hardwired" into the model, because as long as they are *nonlinear* in terms of interval length, they cannot be expressed explicitly in a *linear* model.



Index sets and constant parameters:

$$L := \{i \mid i \text{ is the index of task } T_i\}$$

$$\mathcal{R} := \{R_1, R_2, \dots\} \qquad \text{set of cumulative resources}$$

$$\mathcal{R}_i := \{R_{i1}, R_{i2}, \dots\} \subseteq \mathcal{R},\ i \in L \qquad \text{set of resources used by task } T_i$$

$$K_i := \{c_i, c_i + 1, \dots, (d_i - r_i)\},\ i \in L \qquad \text{interval lengths for } T_i$$

$$w_{ik} := (k - c_i)/k,\ i \in L,\ k \in K_i \qquad \text{weight of interval of length } k \text{ for } T_i$$

Variables:

$$x_{it} = \begin{cases} 1 & \text{if interval for } T_i \text{ occupies slot } t \\ 0 & \text{otherwise} \end{cases} \qquad i \in L,\ -1^* \le t \le D$$

$$y_{ik} = \begin{cases} 1 & \text{if interval of length } k \text{ is selected for } T_i \\ 0 & \text{otherwise} \end{cases} \qquad i \in L,\ k \in K_i$$

Maximise $\sum_{i \in L} \sum_{k \in K_i} y_{ik} w_{ik}$ subject to the following constraints:

Resource availability:

$$\forall R_m \in \mathcal{R}, \forall\, 0 \le t \le D : \sum_{i \in \{i \mid R_m \in \mathcal{R}_i\}} x_{it} \le \mathrm{cap}(R_m, t) \tag{4}$$

Interval contiguity:

$$\forall i \in L, \forall t \in \{r_i - 1, \dots, d_i - 4\}, \forall l \in \{t + 2, \dots, d_i - 2\} :$$

$$x_{it+1} - x_{it} + x_{il+1} - x_{il} < 2 \tag{5}$$

Interval length:

$$\forall i \in L, \forall t \in \{-1^* \dots, r_i - 1, d_i, \dots, D\} : x_{it} = 0 \tag{6}$$

$$\forall i \in L : \sum_{r_i \le t \le d_i} x_{it} \ge c_i \tag{7}$$

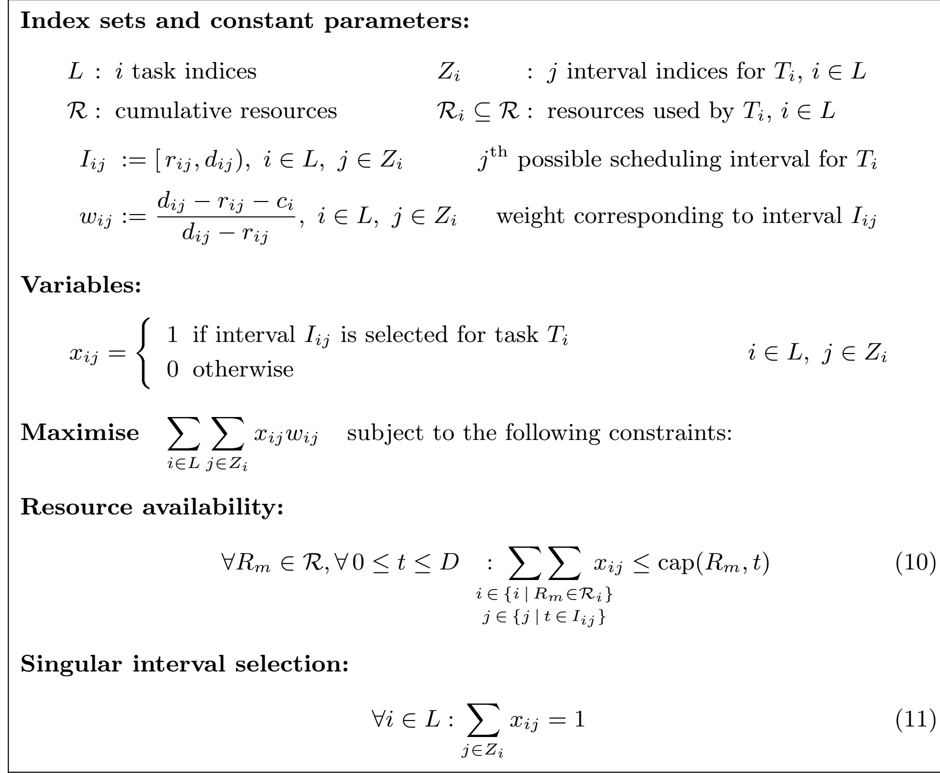
Variable consistency:

$$\forall i \in L \quad : \quad \sum_{r_i \le t \le d_i} x_{it} - \sum_{k \in K_i} k y_{ik} = 0 \tag{8}$$

$$\forall i \in L \quad : \quad \sum_{k \in K_i} y_{ik} = 1 \tag{9}$$

*For technical reasons, the index t of x_{it} can take -1 as its value. Since a task cannot start at time -1, $x_{i,-1} = 0$ by definition.

Figure 3: The slot based ILP model



Index sets and constant parameters:

L : i task indices — Z_i : j interval indices for T_i, $i \in L$

$\mathcal{R}$: cumulative resources — $\mathcal{R}_i \subseteq \mathcal{R}$: resources used by T_i, $i \in L$

$I_{ij} := [r_{ij}, d_{ij}),\ i \in L,\ j \in Z_i$ — j^{th} possible scheduling interval for T_i

$w_{ij} := \dfrac{d_{ij} - r_{ij} - c_i}{d_{ij} - r_{ij}},\ i \in L,\ j \in Z_i$ — weight corresponding to interval I_{ij}

Variables:

$$x_{ij} = \begin{cases} 1 & \text{if interval } I_{ij} \text{ is selected for task } T_i \\ 0 & \text{otherwise} \end{cases} \qquad i \in L,\ j \in Z_i$$

Maximise $\sum_{i \in L} \sum_{j \in Z_i} x_{ij} w_{ij}$ subject to the following constraints:

Resource availability:

$$\forall R_m \in \mathcal{R}, \forall\, 0 \leq t \leq D \quad : \sum_{\substack{i \in \{i \mid R_m \in \mathcal{R}_i\} \\ j \in \{j \mid t \in I_{ij}\}}} \sum x_{ij} \leq \text{cap}(R_m, t) \tag{10}$$

Singular interval selection:

$$\forall i \in L : \sum_{j \in Z_i} x_{ij} = 1 \tag{11}$$

Figure 4: The interval based ILP model

that exactly one out of all the variables belonging to a task is assigned the value 1, in addition to satisfying all the resource and temporal constraints. The complete model is shown in Fig. 4.

The constraints defined by (10) ensure that the time dependent resource capacities are never exceeded. For each particular resource and time slot, we select all the possible scheduling intervals containing this slot of all the tasks using the given resource. The sum of these 0/1 values equals the actual resource usage of any specific solution, therefore it must not be greater than the corresponding capacity. Constraint (11) encodes the requirement that exactly one interval is chosen for each task.

3.4 Jobs as Groups of Tasks

In addition to maintenance tasks, the application domain also operates with the concept of *jobs*. Regular aircraft maintenance consists of independent jobs (e.g. 56-DSI, an inspection scheduled about every 56 days), which are built up from smaller, interrelated tasks (e.g. remove wings to give access to the engine). When

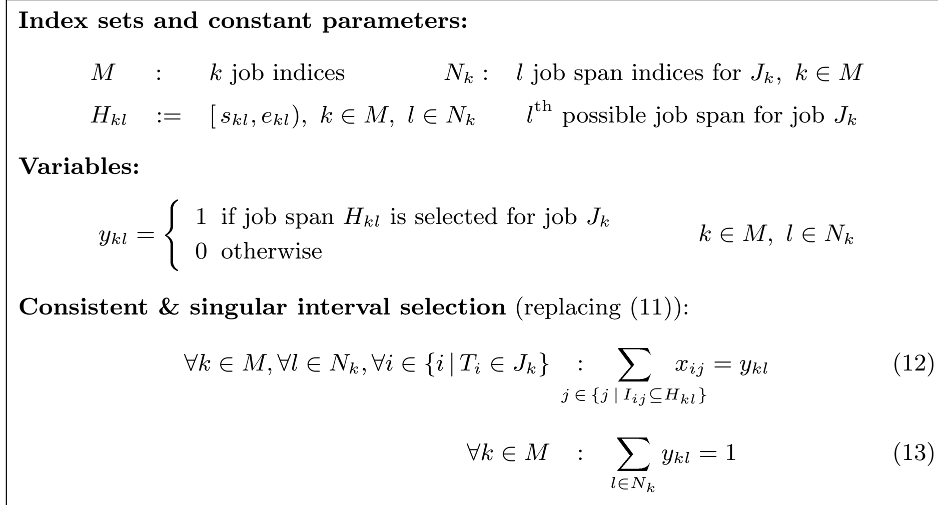


Figure 5: Extending the interval model with jobs

this is translated into a graph, each job constitutes a connected subgraph of the entire task graph, and slack distribution can be performed on these subgraphs independently. In addition, maintenance can only be performed while the aircraft is on the ground, and a job cannot be interrupted with a flight mission (for example because the aircraft is not yet completely reassembled). Assuming that mission planning always precedes maintenance scheduling, the interval based model must be extended to avoid scheduling jobs during and across missions.

Let us introduce the notion of jobs, denoted by J_k, which are sets of T_i tasks, and the concept of *job spans,* those time intervals within which jobs can be performed. A job span is denoted by $H_{kl} = [\,s_{kl}, e_{kl})$, and either *all* or *none* of the tasks constituting the job must be completed in it.

The overall algorithm is then modified as follows. Slack distribution described in Sect. 2.3 is executed for each job (i.e. a connected graph) and *for each job span separately,* assigning a separate scheduling range to each task of the job within each span. Then each of these scheduling ranges is used to generate possible scheduling intervals according to (2), with the added notational complexity that now there is more than one scheduling range per task. The remaining problem is again a singular interval selection for each task, but this time we must also ensure that for all tasks of a single job we select intervals from the same job span.

The extension of the model is shown on Fig. 5 (only new or modified elements are listed). For each job J_k, it introduces a second set of boolean variables, y_{kl}, one for each job span, which is 1 if and only if the corresponding span contains all tasks of the J_k in the resulting schedule.

Equation (11) is replaced with two new equations. Constraint (12) encodes the requirement that all the tasks of a job must be executed in the same job span. The

equation specifies that a particular job span is selected ($y_{kl} = 1$) if and only if all the tasks constituting the job have exactly one interval selected within that job span. Finally, (13) ensures that exactly one job span is selected for each job.

An obvious special case of the extended model is when there is exactly one job, containing all the tasks, and there is exactly one job span. Then the extended model behaves exactly like the simple interval based model on Fig. 4.

3.5 Efficiency Considerations

To avoid the explosion of the number of variables as the number of tasks and the size of the scheduling ranges ($d_i - r_i$) increase, we decided to limit the number of possible scheduling intervals per task. The algorithm we created to determine the intervals is as follows. For each task:

1. The number of intervals per job span is set to be proportional to the size of the span.
2. For each span, generate a set of possible starting points, at which all eventually created intervals will start. The number of points is chosen by keeping two goals in mind:
   - the number of intervals is not less than two per point;
   - the distance between neighbouring points is not less than the execution time of the task.

   The points themselves are distributed evenly within the job span. These rules help to maintain a healthy balance between the number of choices in the starting point and the length of the scheduling interval. And even though the rules are arbitrary, measurement results summarised in Sect. 4.3 indicate that these limitations do not effect the quality of the solutions significantly: the maximal objective values returned by the solvers using the latter model are not worse than those running on the former.
3. For each starting point, generate the required number of intervals (number intervals in the span divided by the number of points), in gradually increasing length from the minimally required up to a globally fixed multiple of the execution time.

Note that due to rounding and integer division, the number of actually generated intervals may not reach the limit. In our current solution, this remainder capacity is unexploited.

Figure 6 shows a robust schedule for the task graph displayed in Fig. 1, generated using an ILP solver on the defined model. Here we assumed that each task uses exactly one of the two available resources. The intervals corresponding to tasks T_5 and T_7 are shorter than the scheduling ranges in the solution, therefore they are shown in bold. The dotted sections are indicating the subintervals removed to satisfy the constraints.

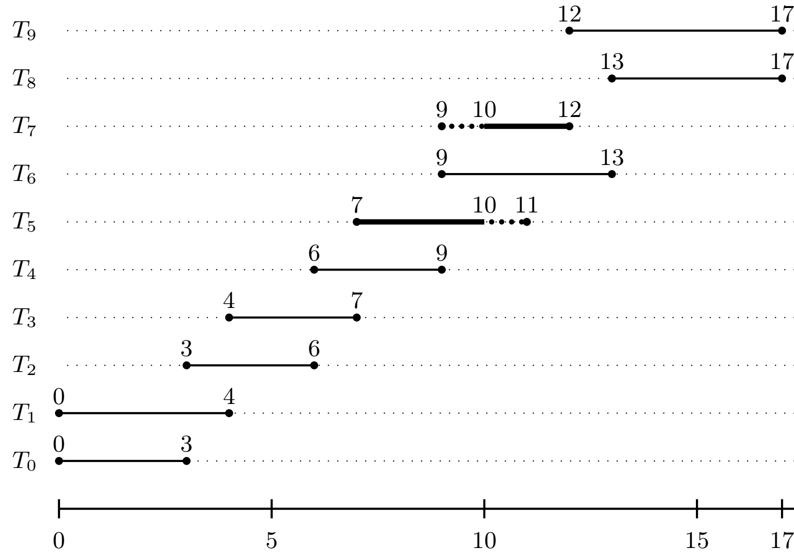


Figure 6: The robust schedule generated with the ILP model

4 Performance Evaluation

In [11], two ILP solvers were tested: a pure ILP called LP_SOLVE[2], a freely available generic linear programming solver [13], and a specialised 0-1 ILP solver targeting pseudo-boolean optimisation problems called PBS [14], which can also handle SAT formulæ. In order to use PBS, the integer constraints in the ILP model were converted to their appropriate pseudo-boolean and conjunctive normal forms. Now we added a second pure ILP solver called GLPK [10] to the list[3], which is also available under a free license.

We have tested the interval based model with real life data take from the software tool mentioned in Sect. 2.1, and verified that the ILP solvers were able to create valid schedules for them in an acceptable time frame. However, the slot based model was not elaborate enough (i.e. no notion of jobs) to handle these scenarios, and also these data represented only a small number of points in the entire problem space. In order to be able to perform a more thorough and systematic evaluation of the models, we turned to random problem generation. Here, our goal was to make the generated problems similar in structure to real life scenarios, so that the evaluation of the former would give us some feel for the performance of the solvers on the latter as well.

[2]We used version 2.0 of LP_SOLVE in our experiments. Since then, newer versions of the software have been released, at the time of writing this article the newest is version 5.5, which might (or might not) perform better on the test data sets.

[3]GLPK version 4.4 was used in the tests. The most recent version at this time is 4.13.

4.1 Expectations

Since the new model does not include constraints which could be encoded as clauses of a SAT formula, we expected a drop in the performance of PBS when moving from the slot based to the interval based model. On the other hand, since the number of constraints has decreased, we hoped that the pure ILP solver will tackle the problem better than before. As for the new GLPK solver, we did not have any previous experience to begin with.

4.2 Problem Generation

The random task graphs used in our experiments were obtained as follows. To generate a *directed acyclic graph* (DAG) with a specific number of tasks, a certain number of *layers* are filled by randomly distributing a number of independent tasks to each layer.[4] Next, we randomly link the edges between tasks in different layers. Finally, tasks are assigned execution times uniformly distributed between [2, 5] secs. A set of resource types $\mathcal{R} = \{R_m\}$, each with a specific capacity is also generated. In our experiments, these resources are distributed uniformly among tasks such that each task is allocated exactly one resource of a certain type. The original resource capacity can also be increased (decreased) as needed. Finally, the graph deadline D is set to $(1 + \mathit{slack}) \cdot p_{max}$, where p_{max} denotes the longest path length through the graph and *slack* is a user-specified value.

4.3 Analysis of Measurement Results

The slot based model has been evaluated with two solvers, LP_SOLVE and PBS. Table 1 summarises the performance of the two solvers given four resource types, each with a capacity of three. The experiments were performed on a 3.2 GHz Pentium 4 processor wit 1 GB of RAM. Graph deadlines are derived using $\mathit{slack} = 1.0$. The table shows the first solution (value of the objective function in the ILP model) returned by both solvers as well as the time taken to do so. (The resolution of the timer was 15 seconds in these experiments.) The solvers were then allowed to improve on their initial solutions up to a time-out period of five minutes and the best solution returned by the solvers after that period is also shown. If a problem is shown to be infeasible by a solver (i.e. it can prove that there is no solution), this fact is denoted by 'Inf.' in the appropriate cell, while a solver time-out without returning any solution is denoted by '—'.
For small numbers of intervals, the solutions returned by LP_SOLVE are superior to PBS at the cost of greater time overhead. For larger numbers of intervals, however, LP_SOLVE was unable to return a solution within the time-out period whereas PBS returned the first solution very quickly.

Now let us turn to the interval based model. It has been evaluated with both solvers used earlier, as well as GLPK, a second pure ILP solver. The results are

[4] The number of tasks per layer is chosen randomly from a specified range, and the number of layers is implicitly determined by the total number of tasks and the number of tasks chosen for each layer.

Table 1: Results of the slot based model with 4 resources of capacity 3, *slack* = 1.0

Tasks	**Scheduling intervals**	**LP_SOLVE**			**PBS**		
		First solution	**Time (secs.)**	**Best solution**	**First solution**	**Time (secs.)**	**Best solution**
∼25	892	12.52	< 15	13.02	2.35	< 15	10.66
∼50	2628	19.41	75	21.26	13.37	< 15	18.92
∼75	2639	18.53	135	20.77	13.44	< 15	17.97
∼100	3091	—	—	—	14.12	< 15	21.83
∼150	7326	—	—	—	30.64	15	33.03

Table 2: Results of the interval based model with 4 resources of cap. 3, *slack* = 1.0

Tasks	**Sched. ints**	**LP_SOLVE**			**PBS**			**GLPK**		
		First sol.	**Time**	**Best sol.**	**First sol.**	**Time**	**Best sol.**	**First sol.**	**Time**	**Best sol.**
∼25	892	13.02	< 5	13.02	11.22	< 5	11.22	13.02	< 5	13.02
∼50	2628	28.00	< 5	28.00	13.82	< 5	16.42	27.99	< 5	28.00
∼100	3091	32.81	< 5	32.81	17.02	10	17.02	32.73	< 5	32.81
∼150	7326	63.4	25	63.4	32.06	15	32.06	63.41	< 5	63.43
∼200	9167	72.77	25	72.82	—	—	—	72.72	10	72.82
∼500	17613	157.78	275	157.78	—	—	—	157.78	15	157.78

summarised in Table 2. The test parameters and conditions were chosen to be identical to the previous tests (in fact, the very same task graphs were used) in order to make comparison possible, only in this case larger problems have been tested as well, since the speedup of the solvers using the new model permitted this increase in size. The time resolution has also been refined to 5 seconds.

The results clearly show that the interval based model suits the taste of the integer linear solvers much better. For all but the smallest problem, the solutions returned by LP_SOLVE are better than with the slot based model, and with much quicker response times. This solver has also been able to cope with problems containing 100–150 tasks, which earlier caused a time-out, and even with problems of 200–500 tasks, which were not even attempted. It is also worth pointing out that the quality of the first and the best solutions differ only minimally (if at all). The results of the GLPK solver are very similar to those of LP_SOLVE, only with smaller run times. (In fact, it would be interesting to observe how GLPK behaves with even larger problem sizes.) On the other hand, the performance of PBS is clearly poorer. Even though the quality of the first solutions is better than before for smaller problems, the quality of the best solutions has diminished. This change for the worse could be explained by the fact that while the slot based model contained a number of SAT encodable constraints, the interval based model does not, and we believe that

Table 3: Effect of *slack* values on solver performance

T#	*slack* = **0.5**				*slack* = **0.8**				*slack* = **1.0**			
	int#	**obj**	**LPS**	**GLPK**	**int#**	**obj**	**LPS**	**GLPK**	**int#**	**obj**	**LPS**	**GLPK**
∼25	436	9.48	0.03	0.13	747	12.52	0.04	0.23	892	13.02	0.05	0.25
∼50	1338	19.66	0.11	0.34	1961	∼ 24.7	0.18	0.53	2628	28	0.27	0.88
∼100	1384	Inf.	0.12	0.28	2277	Inf.	0.31	0.48	3091	32.81	3.72	1.19
∼150	3638	33.2	> 300	2.01	5839	56.65	> 300	3.76	7326	∼ 63.4	> 300	4.3
∼200	4234	Inf.	0.74	1.07	6886	58.79	60.99	2.7	9167	72.82	> 300	12.9
∼500	8582	Inf.	1.99	2.38	14257	∼ 126.3	> 300	30.32	17613	157.78	> 300	17.06

the main strength of PBS, which gave it an edge over LP_SOLVE, lies in the SAT solver core. Since it has not been able to exploit this feature with the interval based model, its performance has degraded.

In [11] we presented a table which emphasised the effect of increasing *slack* values on solver performance. It showed that when *slack* was increased, a larger number of possible scheduling intervals was generated, which in turn caused a larger search space and thus more time-outs. On the other hand, the robustness of the schedules improved where the solvers finished in time. Table 3 shows a similar data set for the interval based model, including results for the two ILP solvers. Since the objective values returned by LP_SOLVE and GLPK were always very close (and *almost* always equal), the table includes a single set of objective values, i.e. those of the best solutions found by both solvers. (A $\sim$ sign denotes where there was a minor difference in the values.) Two further columns per *slack* value show the *total run time* of the two solvers (i.e. the time required to be able to tell that the found solution is indeed the best).
It is interesting to see that there is a jump in the time values of LP_SOLVE, where it was not able to finish within the time limit any more. Nonetheless, it always found a solution[5], which was not even worse than the best solution found by GLPK. However, the time results of GLPK are convincing, it appears that GLPK scales well with the problem size.

5 Conclusions

This paper has addressed the problem of generating robust task schedules under explicit deadline constraints and proposed a new ILP-based solution. In addition to an earlier model of ours, we formulated a second ILP model whose solution maximises the temporal flexibility of the overall task schedule. This model was solved using three integer solvers LP_SOLVE, PBS and GLPK that use widely varying solution techniques. Our experiments show that while LP_SOLVE provides superior solutions for the smallest problems, it is outperformed by GLPK both in

[5] ILP searches can be considered *anytime algorithms* for practical purposes, knowing that they use the *branch-and-bound* algorithm, and assuming that the search tree is interspersed with solutions, which is apparently true our case.

speed and scalability. The SAT based PBS solver finished poorly in our tests. We believe this is because the strength of this solver lies in the SAT solver core, but our ILP model did not contain SAT encodable constraints.

5.1 Future Work

The tests clearly showed that even with the new model, the performance of the solvers degrades drastically as the problem size increases. To face the issue of scalability, we are experimenting with generating solutions in multiple passes, using a technique we call *rolling horizon*. First, scheduling ranges are determined as usual. The idea is then to generate a robust schedule for a relatively short period of the entire planning horizon in each pass, giving greater flexibility to the tasks scheduled at the end of this period, i.e. with the interval weights defined in (3) being modified to be monotonous in r_{ij}. Then in the next pass, the scheduling intervals selected for the trailing part (i.e. part of the output of the first pass) are used as new, reduced scheduling ranges (i.e. as input of the second pass). The second pass will then finalise these tasks by selecting a subinterval of these reduced ranges. This way each pass has the ability to slightly modify the decisions made in the previous pass near the "seams" without breaking any of the already satisfied temporal constraints.

The advantage of this "divide and conquer" approach is that it could help to keep the complexity of the problem linear in the number of tasks. When the ILP models are applied to the entire problem, the increase in run times is steeper than linear as shown in Sect. 4.3. By using a rolling horizon, however, the number of tasks per pass can be kept constant.

Acknowledgements

The work presented here has been completed at the *Institute for Software Integrated Systems*[6], Vanderbilt University in Nashville, TN in 2004. The authors would like to thank the institute, especially *Chris van Buskirk, Gabor Karsai* and *Himanshu Neema* for providing the context, application domain and the necessary resources for the research, and also for giving important ideas, scrutinising our claims and explanations.

Special thanks to *Péter Szeredi* at the *Department of Computer Science and Information Theory*, Budapest University of Technology and Economics for his valuable comments on this paper.

References

[1] Autonomous Negotiating Teams (ANT) Projects. `http://www.dsic-web.net/ito/programs/ant/projects.html`, 2001.

[6] ISIS Web address: `http://www.isis.vanderbilt.edu`

[2] Brucker, Peter. *Scheduling Algorithms.* Springer Verlag, Berlin, fourth edition, 2004.

[3] Bruynooghe, M. Enhancing a search algorithm to perform intelligent backtracking. *Theory and Practice of Logic Programming*, 4(03):371–380, 2004.

[4] Davenport, A.J., Gefflot, C., and Beck, J.C. Slack-based techniques for robust schedules. In *Proc. of the Sixth European Conf. on Planning (ECP-2001)*, Toledo, Spain, September 2001.

[5] Dean, T. and Boddy, M. An analysis of time-dependent planning. *Proceeding of Seventh National Conference on Artificial Intelligence AAAI*, 92:49–54, 1988.

[6] Di Natale, Marco and Stankovic, John A. Dynamic end-to-end guarantees in distributed real-time systems. In *Proc. of IEEE Real-Time Systems Symp.*, pages 216–227, San Juan, Puerto Rico, December 1994.

[7] Drummond, Mark, Bresina, John, and Swanson, Keith. Just-in-case scheduling. In *Proc. of the Twelfth Conf. on Artificial Intelligence (AAAI)*, pages 1098–1104, 1994.

[8] Erlebach, Thomas and Spieksma, Frits C. R. Interval selection: applications, algorithms, and lower bounds. *J. Algorithms*, 46(1):27–53, 2003.

[9] Gao, H. Building robust schedules using temporal protection—an empirical study of constraint-based scheduling under machine failure uncertainity. Master's thesis, Univ. of Toronto, Toronto, Canada, 1995.

[10] GLPK (GNU Linear Programming Toolkit). `http://www.gnu.org/software/glpk/glpk.html`, 2006.

[11] Kandasamy, Nagarajan, Hanak, David, Neema, Himanshu, van Buskirk, Chris, and Karsai, Gabor. Synthesis of robust task schedules for minimum disruption repair. In *Proc. of IEEE International Conference on Systems, Man and Cybernetics (SMC'04)*, pages 5056–5061, The Hague, The Netherlands, October 2004.

[12] Kao, B. and Garcia-Molina, H. Deadline assignment in a distributed soft real-time system. 8(12):1268–1274, December 1997.

[13] LP_SOLVE: A Mixed Integer Linear Programming (MILP) solver. `http://tech.groups.yahoo.com/group/lp_solve`, 2006.

[14] Markov, Igor L., Sakallah, Karem A., Ramani, Arathi, and Aloul, Fadi A. Generic ILP versus specialized 0-1 ILP. In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD'02)*, pages 450–457, San Jose, California, November 2002.

[15] Müller, Tobias. *Constraint Propagation in Mozart.* Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.

[16] Smith, Stephen. OPIS: A methodology and architecture for reactive scheduling. In Zweben, M. and Fox, M., editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.

[17] The MICANTS Project: Model Integrated Computing and Autonomous Negotiating Teams for Autonomous Logistics. `http://www.isis.vanderbilt.edu/Projects/micants/micants.htm`, 2004.

[18] The Mozart Programming System. `http://www.mozart-oz.org`, 2006.

[19] Tsukada, T. K. and Shin, K. G. PRIAM: Polite rescheduler for intelligent automated manufacturing. 12(2):235–245, April 1996.

[20] Würtz, Jörg. Constraint-based scheduling in Oz. In Zimmermann, U., Derigs, U., Gaul, W., Möhrig, R., and Schuster, K.-P., editors, *Operations Research Proceedings 1996*, pages 218–223. Springer-Verlag, Berlin, Heidelberg, New York, 1997. Selected Papers of the Symposium on Operations Research (SOR 96), Braunschweig, Germany, September 3–6, 1996.