2004 IEEE International Conference on Systems, Man and Cybernetics

Synthesis of Robust Task Schedules for Minimum Disruption Repair†

Nagarajan Kandasamy, David Hanak, Chris van Buskirk, Himanshu Neema, and Gabor Karsai

Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN 37203, U.S.A
{nkandasa, dhanak, vbuskirk, himanshu, gabor}@isis.vanderbilt.edu

Abstract: *An off-line scheduling algorithm considers resource, precedence, and synchronization 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). An ILP model, whose solution maximizes the temporal flexibility of the overall task schedule, is formulated. Two different ILP solvers are used to solve this model and their performance compared. For large task graphs, an efficient approximate method is presented and its performance evaluated.*

Keywords: Robust scheduling, slack-based scheduling, integer linear programming.

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 synchronization 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.

This paper addresses the synthesis of robust task schedules using a slack-based approach. We develop a method to construct schedules where individual tasks retain some *temporal flexibility* in the form of slack while satisfying their timing requirements. Therefore, some execution disruptions can be absorbed by the schedule without requiring repair or rescheduling.

There are two general approaches to dealing with schedule disruptions. *Reactive methods* recover from the disruption after it happens, and aim to repair the original schedule in least-disruptive fashion [9] [10]. The authors of [4] precompute a set of contingency schedules and use the one most suited to the prevailing operating conditions. *Proactive methods*, including the one proposed in this paper, construct schedules that can absorb some disruptions without the need for rescheduling. In previously proposed slack-based methods [3] [6], some slack, corresponding to the expected repair time of the resource(s) used by a task, is added to the task execution time prior to scheduling. Standard techniques are then used to generate a robust schedule at the expense of increasing its makespan. On the other hand, this paper assumes tasks with explicit deadline and resource requirements.

Given a task graph with deadline constraints, we address the problem of synthesizing a robust schedule that maximizes the slack added to individual tasks while satisfying their timing requirements. We first discuss how the end-to-end graph deadline is distributed to individual tasks to generate possible scheduling ranges for them. We then present a technique based on integer linear programming (ILP) to select a valid scheduling range for each task such that the temporal flexibility of the overall schedule is maximized. We formulate the ILP model and present experimental results evaluating the performance of two ILP solvers having very different solution methods. Finally, for large task graphs, an approximate or greedy method is proposed and its performance evaluated.

The rest of this paper is organized as follows. Section 2 presents the task model and discusses the deadline distribution algorithm. Section 3 formulates the ILP model for robust schedule generation while Section 4 presents experimental results for two ILP solvers. An approximate method for large graphs is proposed and evaluated in Section 5. We conclude the paper with a discussion on future work in Section 6.

†0-7803-8566-7/04/$20.00 © 2004 IEEE.

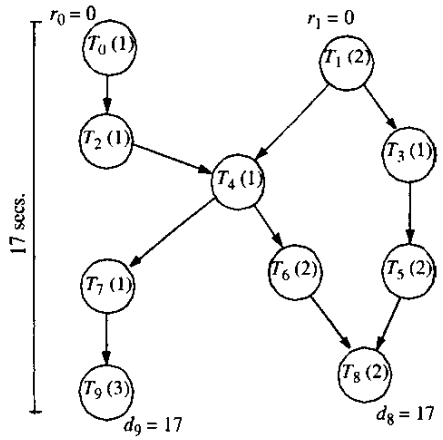

Fig. 1 : An example task graph G with end-to-end deadlines

2 Preliminaries

This section discusses the task model, sources of slack in a task schedule, and the slack distribution algorithm.

Modeling Assumptions

Fig. 1 shows a directed acyclic graph G modeling task interaction. Tasks are non-preemptive and have resource, precedence, and synchronization requirements. The graph comprises vertices and edges representing tasks and precedence constraints, respectively. Each vertex is labeled $T_i(c_i)$, 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 $\{R_m\}$ for its execution where R_m denotes a resource of type m. Also, for each resource R_m, its available capacity is given by $cap(R_m)$.

Scheduling is a mapping of tasks on to resources such that the specified precedence and deadline constraints are satisfied. The desired result is a feasible schedule specifying the release time for each task T_i. It is also necessary to introduce some slack in this schedule to improve its robustness to execution uncertainties, and in many cases, the necessary slack may be obtained by appropriately distributing the end-to-end graph deadline among tasks.

Assume that tasks T_0 and T_1 start at 0 secs., and that G must meet a deadline of 17 secs., i.e., T_8 and T_9 must finish before 17 secs. Note, however, that the longest path $T_0T_2T_4T_6T_8$ through G is only 7 secs. long. This implies that a slack of 17 - 7 = 10 secs. 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 deadline among tasks such that the slack added to each intermediate task is maximized. 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 task deadline, repectively.

Deadline Distribution

Initially, only entry and exit tasks having no predecessors and successors, respectively, have their release times and deadlines fixed. In the *deadline assignment* problem, the graph deadline must be distributed over each intermediate task such that all tasks are feasibly scheduled on their respective resources. Deadline assignment is NP-complete and various heuristics have been proposed to solve it. We use the approach proposed in [8] to maximize 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 favorably with other heuristics [7].

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

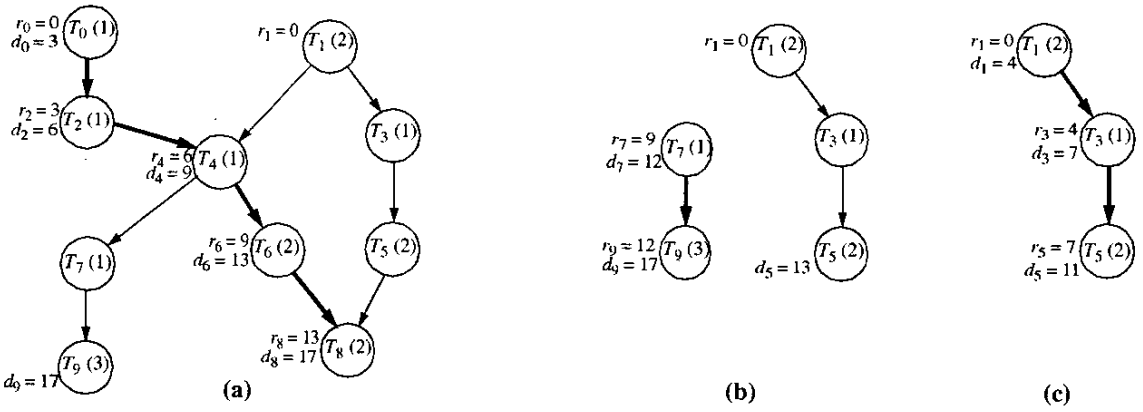

Fig. 2 : (a)-(c) Steps corresponding to the deadline assignment algorithm in [8]; the selected paths are shown as bold edges

$slack_q = D_q - \sum c_i$ where D_q is the end-to-end deadline of $path_q$ and c_i the execution time of task T_i along this path. The distribution heuristic in [8] maximizes the minimum slack added to each T_i along $path_q$ by dividing $slack_q$ equally among tasks. During each iteration through G, $path_q$ minimizing $slack_q/n$, where n denotes the number of tasks along $path_q$, is chosen and the corresponding slack 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 secs. and as per the heuristic, a slack of $(17-7)/5 = 2$ sec. is distributed to each task. Once their release times and deadlines are fixed, these tasks are removed from the graph. Path T_7T_9 in Fig. 2(b) is then chosen and a slack of $\lfloor (8-3)/2 \rfloor = 2$ is added to each task. (Any remaining slack could be distributed to tasks with longer execution times.) Fig. 2(c) shows the final path $T_1T_3T_5$ and the scheduling ranges for the corresponding tasks.

3 Robust Schedule Synthesis

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 on to a limited number of resources, these scheduling ranges must be modified appropriately to account for resource contention during task execution; we adapt concepts from interval scheduling [5] to solve this problem.

The scheduling range for T_i is first decomposed into a number of overlapping intervals $\{I_{ij}\}$. Each I_{ij}, corresponding to the j^{th} possible scheduling interval for T_i, spans $[r_{ij}, d_{ij})$ where r_{ij} and d_{ij} may assume values such that $r_i \le r_{ij} \le d_i - c_i$ and $r_i + c_i \le d_{ij} \le d_i$. Also, I_{ij} is assigned a weight $(d_{ij} - r_{ij} - c_i)/(d_{ij} - r_{ij})$ denoting the scheduling flexibility within that interval in terms of available 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: (1) at any point in the schedule, the overlapping task intervals do not consume more than the number of available resources and (2) the sum of the interval weights is maximized.

Fig.3 shows an ILP model for the interval selection problem whose solution maximizes the sum of interval weights while satisfying a set of linear constraints. The model assumes that while T_i may use multiple resource types, it is allocated exactly one resource from each type. This assumption, however, may be relaxed quite easily.

A schedule of length D (equal to the graph deadline) comprises execution slots of unit length. A feasible solution assigns tasks to these slots such that the following constraints are met.

Constants:

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

K_i := $\{c_i, c_i + 1, \ldots, (d_i - r_i)\}$ (Interval lengths for T_i)

$w_{ik} = \frac{k - c_i}{k}, i \in L, k \in K_i$ (Weight corresponding to an interval of length k for T_i)

Variables:

$$x_{ij} = \begin{cases} 1 & \text{if interval for } T_i \text{ occupies slot } j \\ 0 & \text{if interval for } T_i \text{ does not occupy slot } j \end{cases}$$

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

Maximize $\sum_{i \in L} \sum_{k \in K_i} y_{ik} w_{ik}$ **subject to:**

Resource availability:

$$\forall R_m, \forall j \in \{0, \ldots, D\}: \sum_{i \in \{i \mid T_i \text{ uses } R_m\}} x_{ij} \le cap(R_m)$$

Interval contiguity:

$$\forall i \in L, \forall j \in \{r_i - 1, \ldots, d_i - 4\}, \forall l \in \{j+2, \ldots, d_i - 2\}: \quad x_{ij+1} - x_{ij} + x_{il+1} - x_{il} < 2$$

Interval duration:

$$\forall i \in L, \forall j \in \{0, \ldots, r_i - 1, d_i, \ldots, D\}: \; x_{ij} = 0$$

$$\forall i \in L: \sum_{j \in \{r_i, \ldots, d_i\}} x_{ij} \ge c_i$$

Interval binding:

$$\forall i \in L: \left(\sum_{j \in \{r_i, \ldots, d_i\}} x_{ij} \right) - \left(\sum_{k \in K_i} k y_{ik} \right) = 0$$

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

Fig. 3 : The ILP model for interval selection

1. *Resource capacity*: For each resource type R_m, the capacity constraints ensure that overlapping task intervals do not consume more than the available resources.

2. *Interval contiguity*: Since non-preemptive tasks are assumed, the corresponding scheduling intervals must be contiguous. Therefore, this constraint ensures that a valid schedule comprises only those task intervals spanning contiguous execution slots. We use a simple example to show that these constraints ensure interval contiguity. Assume a non-contiguous interval with "holes", and let $j+1$ be the

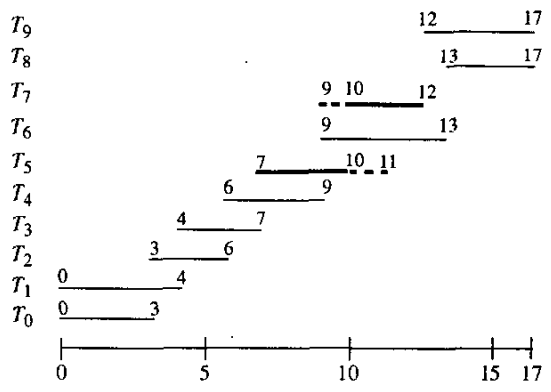

Fig. 4 : The robust schedule generated for the task graph in Fig. 1 by the ILP method given two available resources; each task uses exactly one resource

index of its first slot. Also, let l be the index of the last slot of the (first) hole in the interval. Therefore, $x_{ij} = x_{il} = 0$, and $x_{ij+1} = x_{il+1} = 1$, and $x_{ij+1} - x_{ij} + x_{il+1} - x_{il}$ yields 2, which contradicts the constraint.

3. *Interval duration*: For each task T_i, its scheduling interval must be at least as long as its execution time c_i.

4. *Interval binding*: Once a task interval satisfying the above constraints is selected in the schedule, its length is then determined using the interval binding equations. Since interval weights corresponding to each possible length have been precomputed (to linearize the objective function), the obtained interval length is simply used as an index to the appropriate weight value.

Fig. 4 shows a robust schedule for the task graph in Fig. 1, generated using the ILP method. We assume that each task uses exactly one of two available resources. The intervals corresponding to tasks T_5 and T_7 are shown in bold. The dashed lines indicate portions of the original scheduling ranges pruned to satisfy resource constraints.

4 Performance Evaluation

The foregoing ILP model has been solved using two integer solvers based on widely varying solution techniques.

The random task graphs used in our experiments are obtained as follows. To generate a graph with a specific number of tasks, we randomly distribute a number of independent tasks to each graph layer. 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 $\{R_m\}$, 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 $(1 + slack) \cdot p_{\max}$ where $p_{\max}$ denotes the longest path length through the graph and *slack* is a user-specified value.

We solved the model in Fig. 3 using two solvers; LP_SOLVE (abbreviated as LP in Table 2), a freely available generic linear programming solver [12], and PBS, a specialized 0-1 ILP solver targeting pseudo-boolean optimization problems [1]. (To use PBS, the integer constraints in the ILP model were converted to their appropriate pseudo-boolean and conjunctive normal forms).

Table 1 summarizes the performance of the two solvers given four resources types, each with a capacity of three. The experiments were performed on a 3.2 GHz Pentium 4 processor with one Gigabyte of RAM. Graph deadlines are derived using $slack = 1.0$. 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 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 the solvers, it is denoted by "Inf." in the appropriate cell while a solver time-out without returning any solution is denoted by '–'. For small numbers of scheduling intervals, the solu-

Table 1: Experimental results summarizing the performance of LP_SOLVE and PBS with four resource types, each with a maximum capacity of three, and *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: Effect of *slack* values on solver performance

Tasks	*slack* = 0.5			*slack* = 0.8			*slack* = 1.0		
	Scheduling intervals	LP	PBS	Scheduling intervals	LP	PBS	Scheduling intervals	LP	PBS
~25	436	9.48	6.98	747	12.42	10.42	892	13.02	10.66
~50	1338	15.69	14.27	1961	–	17.63	2628	21.26	18.92
~75	1154	Inf.	Inf.	2340	–	18.13	2639	20.77	17.97
~100	1384	Inf.	Inf.	2277	Inf.	Inf.	3091	–	21.83
~150	3638	–	20.36	5834	–	28.49	7326	–	33.03

Procedure GREEDY(α) /* α := Threshold value */
  $S := \emptyset$; /* Set of currently accepted intervals */
  K := Set of all task intervals in non-decreasing right endpoint order;
  for (each interval in K) **begin**
    i := Current interval;
    C_i := Minimum-weight subset of K such that $(S \setminus C_i) \cup \{i\}$ is feasible;
    if ($weight(C_i) \leq \alpha \cdot weight(i)$) $S := (S \setminus C_i) \cup \{i\}$;
  end;
  return S; /* Return the set of selected intervals */

Fig. 5 : The greedy algorithm for interval selection, summarized from [5]

tions returned by LP_SOLVE are superior to PBS at the cost of greater time overhead. For larger numbers of intervals, however, LP_SOLVE is unable to return a solution within the time-out period whereas PBS returns the first solution very quickly.

Table 2 summarizes the effect of increasing *slack* values on solver performance. Clearly, increasing the *slack* value generates larger numbers of possible scheduling intervals for each task in the graph while providing better schedule robustness. Again, LP_SOLVE is superior to PBS for small problems, while for larger ones, PBS finds feasible solutions when LP_SOLVE does not.

The PBS solver is substantially faster than the more generic LP solver since it has been optimized to specifically handle 0-1 ILP models such as ours. Experimental results presented in [1] also support this conclusion.

5 Approximate Method

We discuss and evaluate an approximate or greedy technique for interval selection when the number of intervals is very large number. We use the algorithm proposed in [5] and summarized in Fig. 5

The algorithm GREEDY accepts a parameter α which can assume values within [0, 1]. The set S of selected intervals is initialized and the scheduling intervals of all tasks are sorted in order of non-decreasing endpoint. When the algorithm processes an interval i, it identifies a minimum-weight set $C_i \subseteq S$ of those selected intervals having resource conflicts with i (including any interval in S belonging to the same task as i). The set C_i is called the cheapest conflict set for i, and the interval i could be added to S if those in C_i are dropped from S. Interval i is selected if $weight(C_i) \leq \alpha \cdot weight(i)$, i.e., the total weight of the selected intervals increases by at least $(1-\alpha) \cdot weight(i)$ if i is selected and intervals belonging to C_i are dropped. Determining the conflict set C_i is equivalent to a graph coloring problem and C_i can be efficiently computed in $O(n)$ time where n is the number of intervals. Therefore, the overall complexity of the greedy algorithm in Fig. 5 is $O(n^2)$.

We have evaluated the performance of the greedy algorithm on random task graphs and the results are shown in Table 3. We use *slack* = 1.0 to derive graph deadlines and assume four resource types, each with a capacity of three. The GREEDY algorithm is repeatedly invoked with α assuming values between [0, 1] in increments of 0.1. The table shows the sum of selected interval weights, the percentage of tasks successfully scheduled, and the time taken for the entire run. Note that GREEDY is unable to fully schedule the entire tasks set given the above setup.

6 Discussion

This paper has addressed the problem of generating robust task schedules under explicit deadline constraints and proposed an ILP-based solution. We formulated an ILP model whose solution maximizes the temporal flexibility

Table 3: Experimental results summarizing the performance of the greedy algorithm

Tasks	Intervals	*weight*(*S*)	% tasks scheduled	Time (secs.)
50	2628	24.57	79.63	0.66
100	3091	37.42	81.55	0.76
150	7326	66.55	83.77	2.16
200	9167	91.89	83.5	3.40
500	17613	183.73	76.14	6.56

of the overall task schedule. This model was solved using two integer solvers LP_SOLVE and PBS that use widely varying solution techniques. Our experiments show that while LP_SOLVE provides superior solutions for small problems, PBS is able to quickly find feasible solutions for larger problems that LP_SOLVE cannot solve. An efficient and approximate algorithm to generate robust schedules was also presented and evaluated for large task graphs.

Greedy algorithms for interval selection appear reasonable for very large task graphs, such as those found in some real-world scheduling problems; for example, aircraft maintenance [11]. However, these algorithms may be unable to schedule entire task sets (as indicated by our experiments). Task priorities, if taken into account during interval selection, can improve solution quality by scheduling higher priority tasks over others. Slack distribution strategies taking into account task priorities, execution-time uncertainties associated with individual tasks, and failure rates of critical resources used by the tasks as well as their repair-time distribution can also be investigated. These issues are a focus of ongoing and future work.

Acknowledgements

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-2-0505. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

References

[1] F. A. Aloul, A. Ramani, I. L. Markov, and S. A. Sakallah, "Generic ILP versus Specialized 0-1 ILP: An Update," *IEEE Conf. Computer Aided Design (ICCAD)*, pp. 450-457, 2002.

[2] P. Brucker, *Scheduling Algorithms*, 2nd Edition, Springer Verlag, Berlin, 1998.

[3] A. J. Davenport, C. Gefflot, and J. C. Beck, "Slack-based Techniques for Robust Schedules," *Proc. 6th European Conf. Planning*, 2001.

[4] M. Drummond, M. Bresina, and K. Swanson, "Just-in-Case Scheduling," *Proc. 12th Conf. Artificial Intelligence (AAAI)*, pp. 1098-1104, 1994.

[5] T. Erlebach and F. C. R. Spieksma, *Interval Selection: Applications, Algorithms, and Lower Bounds*, Tech. Report No. 152, Computer Engineering & Networks lab., Swiss Federal Institute of Technology, Zurich, October 2002.

[6] H. Gao, *Building Robust Schedules using Temporal Protection-An Empirical Study of Constraint-based Scheduling under Machine Failure Uncertainty*, M. S. Thesis, University of Toronto, 1995.

[7] B. Kao and H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System," *IEEE Trans. Parallel and Distributed Syst.*, vol. 8, no. 12, pp. 1268-1274, Dec. 1997.

[8] M. D. Natale and J. A. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real-Time Systems," *Proc. Real-Time Systems Symp.*, pp. 216-227, 1994.

[9] S. Smith, "OPIS: A Methodology and Architecture for Reactive Scheduling," *Intelligent Scheduling*, (Eds. M. Zweben and M. S. Fox), pp. 29-66, Morgan Kaufmann, San Mateo, CA, 1994.

[10] T. K. Tsukada and K. G. Shin, "PRIAM: Polite Rescheduler for Intelligent Automated Manufacturing," *IEEE Trans. Robotics & Automation*, vol. 12, no. 2, pp. 235-245, April 1996.

[11] MAPLANT, Institute for Software Integrated Systems, Vanderbilt University, http://www.isis.vanderbilt.edu/maplant

[12] LP_SOLVE, ftp://ftp.es.ele.tue.nl/pub/lp_solve