A Uniprocessor Scheduling Policy for Non-Preemptive Task Sets with Precedence and Temporal Constraints

Matthew C. Gombolay and Julie A. Shah
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
gombolay@csail.mit.edu, julie_a_shah@csail.mit.edu

We present an idling, dynamic priority scheduling policy for non-preemptive task sets with precedence, wait constraints, and deadline constraints. The policy operates on a well-formed task model where tasks are related through a hierarchical temporal constraint structure found in many real-world applications. In general, the problem of sequencing according to both upperbound and lowerbound temporal constraints requires an idling scheduling policy and is known to be NP-complete. However, we show through empirical evaluation that, for a given task set, our polynomial-time scheduling policy is able to sequence the tasks such that the overall duration required to execute the task set, the makespan, is within a few percent of the theoretical, lowerbound makespan.

I. Introduction

The sequencing and scheduling of tasks according to precedence and temporal constraints is a challenging problem with important applications, for example, in autonomous tasking of unmanned aerial[14] and underwater vehicles,[3] scheduling of factory operations,[9] and queuing of aircraft on airport runways and taxiways.[13] New uses of robotics for flexible manufacturing are pushing the limits of current state-of-the-art methods in artificial intelligence (AI) and operations research (OR) and are spurring industrial interest in fast methods for sequencing and scheduling.

Methods from the AI[1,15,17] and OR communities[11,16] provide complete search algorithms that require exponential time to compute a solution in the worst case. Practically, the sequencing of fifty or more tasks in the domains of interest often takes a half-hour or more. These methods cannot provide fast re-computation of the schedule in response to on-the-fly disturbances for large, real-world task sets.

In this paper, we provide a dynamic-priority, polynomial-time scheduling policy for task sets with precedence and upper and lowerbound temporal deadlines. Our approach is inspired by large-scale, successful task planning systems in AI that leverage the hierarchical structure of task networks to achieve improvements in computation time.[17] We translate this insight to real-time processor scheduling and present a modified earliest deadline first (EDF) policy called Column-Restricted EDF that leverages the hierarchical structure of the task relations found in many real-world applications to generate a schedule with good characteristics: low processor (or agent) idle time and tight makespan, which means that the overall duration required to execute the task set is near-optimal. CR-EDF is not optimal; in general the problem of sequencing according to both upperbound and lowerbound temporal constraints requires an idling scheduling policy and is known to be NP-complete.[10,12] However, we show through empirical evaluation that schedules resulting from the CR-EDF policy are within a few percent of the best possible makespan.

We begin in Section II by presenting a well-formed task model that is hierarchically composed according to a set of primitive combinators. In Sections III and IV, we present our CR-EDF scheduling policy for uniprocessor scheduling of well-formed task models and empirically validate the tightness of the schedules produced relative to the theoretical lowerbound of the scheduler.

II. Well-Formed Task Model

We define a well-formed task model as a task set, τ, that is composed of tasks $\tau_i, i \in \{1, \ldots, n\}$, where each τ_i has an associated execution time, or cost, c_i. The model is well-formed in that tasks are related

through upperbound deadlines and lowerbound wait constraints according to a set of primitive combinators defined below. The resulting task model has a well-defined network structure that we leverage to create an empirically tight scheduling policy. In this paper, we consider non-preemptive task sets and leave the generalization to preemptable tasks for future work.

Definition 1 *Well-Formed Task Model. A well-formed network consists of an epoch and terminus task each with zero cost, and a set of intermediate tasks each with non-zero cost. The epoch serves as a trigger for the release of the network. An upperbound temporal constraint on the makespan, m, may be placed on the execution time from the epoch to terminus. Tasks may also be related through upper and lowerbound temporal constraints using the following set of four primitive combinators.*

II.A. Primitive Combinators

Serial Task Combinator: We define the serial task combinator as the ordering in series of two tasks, τ_i and τ_j, into a super task, τ_k, where the start of task τ_j must follow the end of τ_i after minimum wait time, $\theta_{i,j}$. Similarly, two super tasks, or a task and a super task may be combined with the serial task combinator. Figure 1 shows an illustration of the serial task combinator.

Parallel Task Combinator: We define parallel task combinator as the parallelization of a set of tasks, $\{\tau_i, \ldots, \tau_j\}$, into a super task, τ_k, where each task in $\{\tau_i, \ldots, \tau_j\}$ begins a minimum wait time, $\{\theta_{\{k,i\}}, \ldots, \theta_{\{k,j\}}\}$, respectively, after the start of τ_k. Similarly, each task ends a minimum wait time $\{\theta_{\{i,k\}}, \ldots, \theta_{\{j,k\}}\}$ before the end of τ_k. Tasks and super tasks may be combined with the parallel task combinator. Figure 2 shows an illustration of parallel task combination.

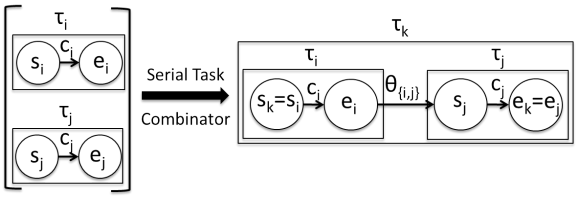

Figure 1. Serial Task Combinator.

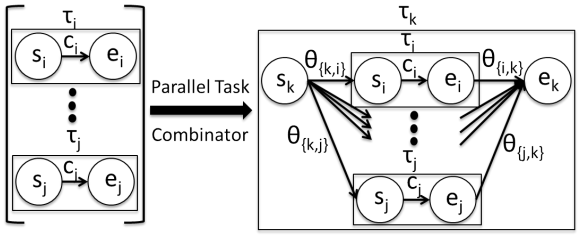

Figure 2. Parallel Task Combinator.

Task-Task Deadline Combinator: We define the task-task deadline combinator, as a constraint, d_i, on the upperbound of the allowable duration between the start and finish time of task or super task, τ_i. Figure 3 shows the graphical illustration of task-task deadline combination. When a task, τ_i, is scheduled with an associated task-task deadline, d_i, that deadline constraint is considered active while $s_i \leq t \leq f_i$ where s_i is the start time of τ_i, t is the current time, and f_i is the finish time of τ_i. In the case that τ_i is not a supertask, the deadline constraint is trivial because non-preemptable tasks execute with a fixed duration, c.

Epoch-Task Deadline Combinator: We define the epoch-task deadline combinator as a constraint, $d_{\{k,i\}}$, on the upperbound of the duration between the start time of a supertask, τ_k, formed by the Parallel Task Combinator, and finish time of a task, τ_i.

Figure 4 shows a graphical description of epoch-task deadline combination. When a super task, τ_k, is scheduled and has an associated epoch-task deadline constraint, $d_{\{k,i\}}$, from the start of τ_k to the end of τ_i, as shown in Figure 4, that epoch-task deadline is considered active while $s_k \leq t \leq f_i$ where s_k is the start time of τ_k, t is the current time, and f_i is the finish time of τ_i.

The task-task and epoch-task deadlines are similar to the latency constraints discussed in[6,7] , where latency is defined as a limit on the upperbound duration relating the start of two tasks. The key difference is that the task-task and epoch-task deadline combinators, instead, limit the upperbound duration between the start of one task and the end of another task.

II.B. Complex Deadline Constraints

As a consequence of combination, one task-task deadline constraint may be nested within another task-task deadline constraint as shown in Figure 5. If a task is involved in one of a set of nested task-task deadlines,

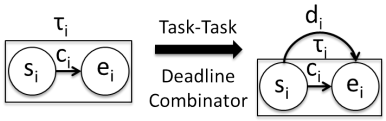

Figure 3. Task-Task Deadline Combinator.

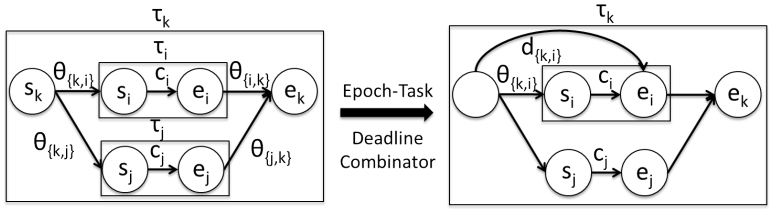

Figure 4. Epoch-Task Deadline Combinator.

then the task is said to be involved in a complex deadline constraint. We define $\{\tau_{CDC}\}$ as the set of tasks with complex deadline constraints. Figure 5 shows two nested task-task deadlines that form complex deadline constraints, where $\{\tau_{CDC}\} = \{\tau_i, \tau_j\}$. Epoch-task deadlines may also be nested to form complex deadline constraints, where each task involved is likewise in the set $\{\tau_{CDC}\}$. To support efficient inferences on the task network structure, we add a restriction on the use of complex deadline constraints as follows: an epoch-task and a task-task deadline cannot be combined to create a complex deadline constraint.

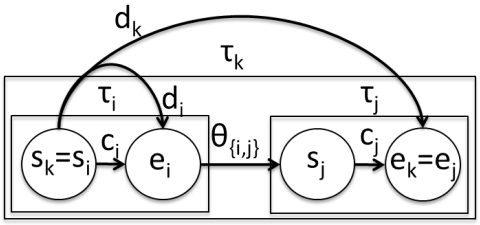

Figure 5. A complex task-task deadline constraint.

II.C. Example Well-Formed Task Model

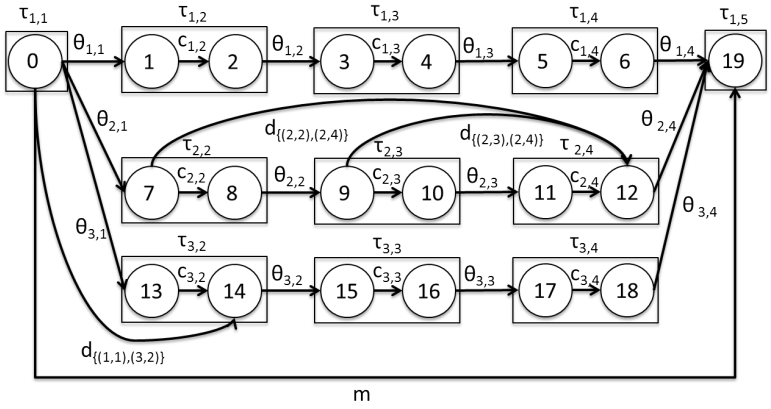

Figure 6. Example of a well-formed, task model.

Figure 6 shows an example of a well-formed task network constructed using the four primitive combinators. This network is composed of eleven tasks: nine tasks related through serial and parallel combination and one epoch and terminus task; the network has one epoch-task deadline and two task-task deadlines. The two task-task deadlines are complex deadline constraints, because one is nested within the other. Here, the set of complex deadline constraints is shown in Equation 1.

$$\{\tau_{CDC}\} = \{\tau_{2,2}, \tau_{2,3}, \tau_{2,4}\} \tag{1}$$

II.D. Notation and Definitions

We adopt the following naming conventions for elements of the well-formed task model. This notation allows us to compactly represent tasks, costs, and waits as indexed matrices, T, C, Θ.

Tasks are identified with subscripts that correspond to row and column. For example, $\tau_{1,2}$ is visually depicted in Figure 6 in the first row and second column of tasks. Likewise, costs $c_{i,j}$, and wait durations $\theta_{i,j}$, are identified by row and column.

Equations 2-4 present T, C, Θ for the example network in Figure 6. Recall that by Definition 1, $c_{1,1}$ and $c_{1,5}$ are zero cost.

$$T = \begin{bmatrix} \tau_{1,1} & \tau_{1,2} & \tau_{1,3} & \tau_{1,4} & \tau_{1,5} \\ 0 & \tau_{2,2} & \tau_{2,3} & \tau_{2,4} & 0 \\ 0 & \tau_{3,2} & \tau_{3,3} & \tau_{3,4} & 0 \end{bmatrix} \tag{2}$$

$$C = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} & c_{1,5} \\ 0 & c_{2,2} & c_{2,3} & c_{2,4} & 0 \\ 0 & c_{3,2} & c_{3,3} & c_{3,4} & 0 \end{bmatrix} \tag{3}$$

$$\Theta = \begin{bmatrix} \theta_{1,1} & \theta_{1,2} & \theta_{1,3} & \theta_{1,4} \\ \theta_{2,1} & \theta_{2,2} & \theta_{2,3} & \theta_{2,4} \\ \theta_{3,1} & \theta_{3,2} & \theta_{3,3} & \theta_{3,4} \end{bmatrix} \tag{4}$$

Task networks with unequal numbers of tasks in rows or columns are encoded using zeros in the matrices to fill the empty positions. The indices for non-zero components may be chosen for favorable computational characteristics; we will discuss potential approaches for this choice in future work.

In the example shown in Figure 6, we note that the corresponding T and C matrices have three rows and five columns; however, the Θ matrix has but four columns. Because wait constraints connect two tasks, we expect networks to have corresponding Θ matrices with one column fewer than the corresponding T and C matrices. For the purposes of notation, when we discuss the number of columns of a network, *ncols*, we refer to the number of columns in T. This convention will be important for our formulations of the upper and lowerbound makespan for a task network in Section V.A, Equations 19-20. The number of rows, *nrows*, is consistent between the three matrices.

Next, we introduce definitions that relate to the scheduling of a well-formed network. These terms will be referred to in Sections III-IV to describe the scheduling policy.

Definition 2 *Release of a Task. During the scheduling of a well-formed task set, we say a task or supertask, τ_i, is released when all temporal and precedence constraints for the task start, s_i, have been satisfied.*

Definition 3 *Explicit Task Deadline. An explicit deadline is an upperbound temporal constraint added through the use of an epoch/task-task deadline combinator. All tasks involved in the epoch/task-task combination are said to be constrained by the explicit deadline, which means the tasks' deadlines may be affected by the execution time of predecessor tasks involved in the deadline constraint.*

Definition 4 *Implicit Task Deadline. All tasks not constrained by an explicit deadline are said to have implicit deadlines. This means that the maximum allowable start and finish times of the tasks are implicitly constrained by the other costs, waits, and deadlines in the network. Implicit deadlines may be computed through an all-pairs-shortest-path (APSP) computation on a temporal distance graph.*[8]

Definition 5 *Free Task. A free task is defined as a task that either (1) is not constrained by an explicit task deadline, (2) is the epoch task, or (3) begins a task-task deadline and does not share any explicit deadline constraint with a preceding task.*

The intuitive explanation is that a scheduler may assign the start time of a free, released task anytime within the implicit deadline associated with the start of the task, and the task's deadline is not affected by the execution time of predecessor tasks. The set of free tasks $\{\tau_{free}\}$ in the Figure 6 example includes: $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{1,4}, \tau_{1,5}, \tau_{2,2}, \tau_{3,3}, \tau_{3,4}$.

Definition 6 *Embedded Task. An embedded task is a task that does not satisfy the criteria for a free task and is therefore constrained by an explicit deadline.*

The intuitive explanation is that the allowable start time of an embedded task is affected by the execution times of predecessor tasks involved in the deadline constraint. The set of embedded tasks $\{\tau_{embedded}\}$ in the Figure 6 network includes: $\tau_{2,3}, \tau_{2,4}, \tau_{3,2}$.

Definition 7 *Active Task. A task that is currently being executed, or a task with an active deadline constraint.*

Consider the well-formed network in Figure 6 as an example. The embedded task $\tau_{2,3}$ becomes active once $\tau_{2,2}$ is complete, because its predecessor task has finished executing. Similarly, $\tau_{3,2}$ becomes an active task as soon as the network is started with $\tau_{1,1}$. Note, however that $\tau_{1,2}$ remains inactive even after $\tau_{1,1}$ is complete, because $\tau_{1,2}$ is a free task.

II.E. Real-World Applications

We can represent many real-world constraints through the four primitive combinators presented in Section II. For example, serial combination may be applied to encode precedence constraints or denote minimum wait times between tasks. Wait constraints arise, for example, in robotic painting where time is necessary for the paint to cure, or in air traffic scheduling, where landing aircraft must be spaced by a minimum separation time.

The parallel combination encodes choice in the valid sequencing. Parallel and serial combinators together form a partially ordered plan. Task-task and epoch-task deadlines also arise naturally in factory operations and airplane scheduling. For example, a factory supervisor may specify that a sequence of tasks should be accomplished in the first shift, or should be grouped together to be executed within a specified window. Deadlines may also encode the amount of time an aircraft can remain in a holding pattern based on fuel considerations.

Next, in Section III, we develop a modified Earliest-Deadline-First (EDF) dynamic-priority, scheduling algorithm to execute non-preemptive, well-formed task sets. We show empirically that this modified EDF strategy achieves execution times within a few percent of the best possible makespan.

III. Motivation for a Modified Earliest-Deadline-First Scheduling Strategy

III.A. Preliminaries

Correct scheduling of the well-formed task model requires the use of a non-preemptive, idling scheduling policy. Using an idling scheduling policy, a released task may either be scheduled immediately or wait to be scheduled even if the processor (or agent) is not busy. The scheduler of a well-formed task set must permit idling to accommodate interactions between minimum waits and task deadlines. Figure 7 presents an example well-formed task network that illustrates the necessity of an idling policy.

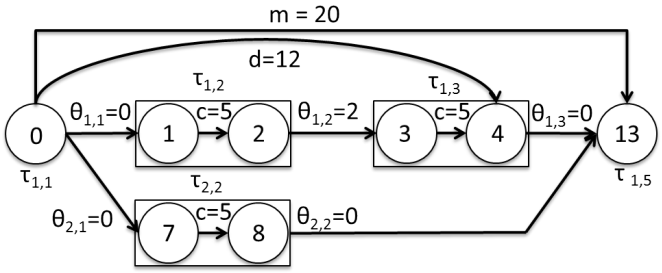

Figure 7. To schedule this network, we need an idling scheduling policy.

The tasks must be sequenced in the order $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{2,2}, \tau_{1,5}$ to satisfy the deadline constraint, $d = 12$, and the makespan constraint $m = 20$. Note this requires an idling period of $\theta_{1,2} = 2$ between $\tau_{1,2}$ and $\tau_{1,3}$. This idling period results because $\tau_{2,2}$ cannot be inserted between $\tau_{1,2}$ and $\tau_{1,3}$ without violating $d = 12$.

The general problem of finding a feasible schedule in an idling and non-preemptive context is known to be NP-Complete.[10,12] An exhaustive search leads to $n!$ different schedules in the worst case.

Prior work has analyzed the optimality of an Earliest-Deadline First (EDF) scheduling strategy and shown that it is optimal for uniprocessor preemptive scheduling of periodic/aperiodic tasks.[2] This work has been extended to systems with precedence, deadline, and latency constraints.[4–7,18] Nonetheless, non-preemptive EDF strategies under these conditions are known to be suboptimal.[12]

We introduce a modified EDF policy called Column-Restricted EDF (CR-EDF) that, although also suboptimal, leverages the structure of the well-formed network to produce schedules that achieve execution times within a few percent of the best possible makespan. In Section IV, we present CR-EDF in full detail with examples, and in Section V we empirically validate that CR-EDF applied to well-formed task models produces schedules that are within a few percent of the theoretical lowerbound for the makespan returned by the scheduler.

In the following example, we motivate the use of our modified EDF policy. We provide an example task network that illustrates a situation that arises frequently in real-world problems, and show that an idling CR-EDF policy can schedule the task set where an idling EDF policy cannot.

III.B. Motivating Example for CR-EDF: Considering a Well-Formed Network without Explicit Deadline Constraints.

Consider the task network, shown in Figure 8 with tasks, T, task costs C, and waits Θ as defined in Equations 5-7.

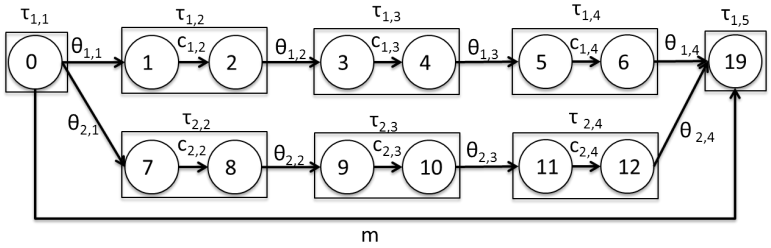

Figure 8. Network that we use to illustrate the value of scheduling with CR-EDF rather than EDF.

$$T = \begin{bmatrix} \tau_{1,1} & \tau_{1,2} & \tau_{1,3} & \tau_{1,4} & \tau_{1,5} \\ 0 & \tau_{2,2} & \tau_{2,3} & \tau_{2,4} & 0 \end{bmatrix} \tag{5}$$

$$C = \begin{bmatrix} 0 & 1 & 1 & 5 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{6}$$

$$\Theta = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{7}$$

In this example, we set an upperbound constraint on the makespan, $m = 10$. Tasks $\tau_{1,2}$ and $\tau_{1,3}$ have implicit deadlines of $t = 5$ and 6 respectively, based on APSP computation.

Execution according to an idling Earliest Deadline First (EDF) policy would execute tasks $\tau_{1,2}$ and $\tau_{1,3}$ before executing $\tau_{2,2}$. This sequencing results in idle time between $\tau_{2,3}$ and $\tau_{2,4}$, as shown in the top timeline in Figure 9. We refer to the behavior where the execution pattern advances further down one task row than another as deadline advancement. Because of this deadline advancement, EDF fails to satisfy the upperbound deadline on the makespan of $m = 10$, thus resulting in an infeasible schedule.

However, if we were not scheduling under EDF, we could insert $\tau_{2,2}$ between $\tau_{1,2}$ and $\tau_{1,3}$ while still meeting $\tau_{1,3}$'s deadline ($t = 5$) even though $\tau_{1,3}$'s deadline is a more urgent deadline than the $\tau_{2,2}$'s deadline ($t = 6$). This altered schedule would still be temporally feasible, and we would still satisfy the makespan, $m = 10$, which we can see in the bottom timeline shown in Figure 9. This altered strategy is a policy that we call Column-Restricted EDF, which we will define next.

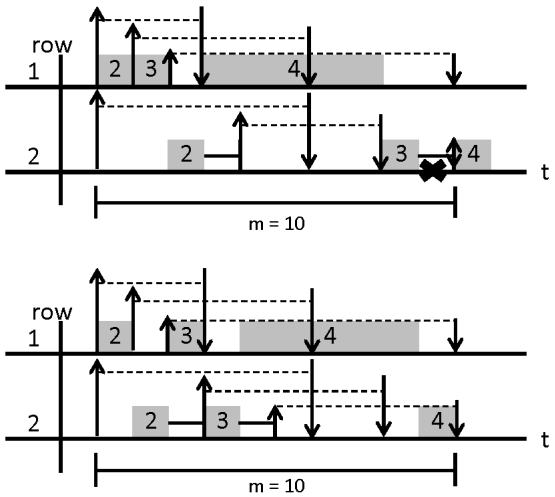

Figure 9. Two timelines of the execution of the network shown in Figure 8. The top timeline shows the execution pattern according to EDF, and the bottom timeline shows the execution pattern according to CR-EDF. Grey bars represent task costs, and the number inside the gray bar corresponds with the column number of that task. The horizontal lines following a gray task bars are the ensuing wait constraints (e.g. after $\tau_{2,2}$ and $\tau_{2,3}$). Upward and downward arrows represent the releases and deadlines of tasks, respectively, which are paired by height and connected with a dashed line. An 'x' on the time-axis represents processor idle time. The execution of the epoch task, $\tau_{1,1}$, where $s_{1,1} = f_{1,1} = 0$ regardless of the scheduling policy, is not shown.

IV. Column-Restricted EDF Explained

Definition 8 *Column-Restricted EDF (CR-EDF). CR-EDF is a scheduling policy for the well-formed task model that maintains a queue of all active tasks and attempts to schedule tasks that have been released, according to deadline priority, by performing an online consistency check.*

The online consistency check described in Definition 8 consists of three conditions. We note that CR-EDF is a just-in-time scheduling policy that concurrently schedules (i.e., assigns a begin time) and begins execution of each task.

1. For an agent to schedule $\tau_{candidate}$ in column j, all free tasks in columns less than j must have been scheduled.

2. We cannot schedule $\tau_{candidate}$ if doing so would result in an active task missing its deadline.

3. An agent cannot execute a complex deadline constraint while that agent is executing another deadline constraint, complex or otherwise.

For Condition 1, we leverage the structure of the well-formed task model and restrict the execution of free tasks such that all free tasks in column j must be executed before a free task in column $j + 1$ is scheduled. This restriction prevents deadline advancements such as those described in Section III.B. Furthermore, free tasks in the column are considered in order of priority by earliest deadline first. Hence, we have a column-restricted EDF policy.

The second condition leverages the structure of our hierarchical task model as well. In short, CR-EDF attempts to schedule tasks during wait constraints that connect free or embedded tasks with other embedded tasks, while ensuring that the over-arching deadlines are satisfied. We evaluate the feasibility of executing $\tau_{candidate}$ while there are active tasks through a check called the *Russian Dolls Test*, which we will define in Section IV.A.

The third condition is necessary to ensure the validity of the Russian Dolls Test. We will explain this more fully in Section IV.A.

IV.A. Russian Dolls Test

Before we can define how the Russian Dolls test works, we must know three pieces of information about $\tau_{candidate}$, and all active tasks, $\{\tau_{active}\}$ (Definition 7). As a mechanism for collecting this information, we will define a new term called a *task group*.

Definition 9 *Task Group. A* task group*, G, is either the set of sub-tasks within a super task that were combined using a deadline combinator (called a deadline task group), or a free task without an associated deadline constraint (called a non-deadline task group). The operator $G(\tau_{i,j})$ returns the set of task(s) in $\tau_{i,j}$'s task group.*

For example, in Figure 6, there are three deadline task groups, and six non-deadline task groups. The deadline task groups are 1) $G(\tau_{2,2}) = \{\tau_{2,2}, \tau_{2,3}, \tau_{2,4}\}$, 2) $G(\tau_{2,3}) = \{\tau_{2,3}, \tau_{2,4}\}$ 3) $G(\tau_{1,1}) = \{\tau_{1,1}, \tau_{3,2}\}$. The six remaining task groups are the tasks in the task network not listed in one of the deadline task groups.

Now, we can define the three parameters necessary to perform the Russian Dolls Test.

- t_{min}, the minimum time required to execute all tasks in a task group. This duration is represented by the APSP lowerbound time difference between the start of the first task and end of the last task within a task group.
- t_{max}, the time available to execute the tasks in the task group. This duration is represented by the APSP upperbound time difference between the start of the first and end of the last task in the task group.
- t_{δ}, the deadline slack time available for the agent to execute a task not in the task group. This duration is equal to the difference between t_{max} and t_{min}.

With these parameters defined, we now describe the Russian Dolls Test.

Definition 10 *Russian Dolls Test. Given an agent with a set of active tasks, the Russian Dolls Test determines whether or not a candidate task's task group can be nested within the task groups of the embedded tasks. If we order the set of all active task groups and the candidate's task group, according to size (t_{max}), the slack, t_{δ}, for the group with the i^{th}-largest t_{max} must be greater than the group with the $(i-1)^{th}$-largest t_{max}. This process is similar to how toy Russian Dolls can be nested inside of each other. If we order Russian Dolls by size, a larger doll must be able to nest the next smaller doll within its cavity.*

We will walk through three examples to illustrate the Russian Dolls Test. First, we will consider two scenarios where $\tau_{candidate}$ passes the Russian Doll Test. Second, we will consider a scenario where $\tau_{candidate}$ fails the Russian Dolls Test.

Consider the illustrations in Figures 10 and 11. In each Figure, there are three task groups: one group for the candidate task and two task groups for active, embedded tasks. All tasks have a cost of 1, and there is a wait constraint between each task equal to 1. The values of t_{max}, t_{min}, and t_{δ} for the left and right examples in Figure 10 and 11 are shown in Tables 1 and 2, respectively.

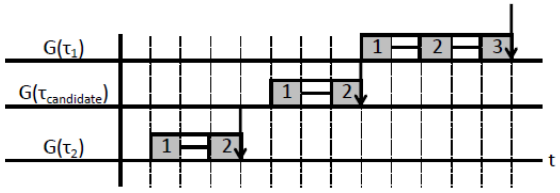

Figure 10. Visual description of a Russian Dolls Test for $\tau_{candidate}$. Numbers inside of the gray boxes correspond to the index of the task in the task group. Here, $\tau_{candidate}$ is a member of a task group that has a deadline constraint. $\tau_{candidate}$ passes the Russian Dolls Test. Parameters of the test are shown in Table 1.

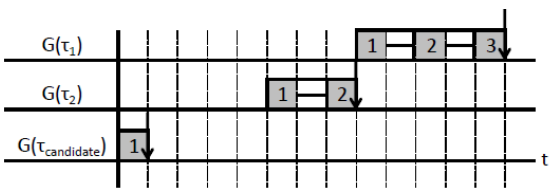

Figure 11. Visual description of a Russian Dolls Test for $\tau_{candidate}$. Numbers inside of the gray boxes correspond to the index of the task in the task group. Here, $\tau_{candidate}$ is a free task without a deadline constraint. $\tau_{candidate}$ passes the Russian Dolls Test. Parameters of the test are shown in Table 2.

In Figure 10, we can see that $\tau_{candidate}$ passes the Russian Doll test because we can nest $G(\tau_{candidate})$ in the slack of $G(\tau_1)$, and we can nest $G(\tau_2)$ in the slack of $G(\tau_{candidate})$. Likewise, in Table 1, we can order

Task Group	t_{max}	t_{min}	t_δ
$G(\tau_1)$	13	5	8
$G(\tau_{candidate})$	8	3	5
$G(\tau_2)$	4	3	1

Table 1. Table of values for an example of a Russian Dolls Test corresponding to the leftmost timeline in Figure 10.

Task Group	t_{max}	t_{min}	t_δ
$G(\tau_1)$	13	5	8
$G(\tau_2)$	8	3	5
$G(\tau_{candidate})$	1	1	0

Table 2. Table of values for an example of a Russian Dolls Test corresponding to the rightmost timeline in Figure 11.

the groups according to t_{max} and find that t_δ for a given group is greater than or equal to t_{max} of a group with a smaller t_{max}.

In Figure 11, we can see that $\tau_{candidate}$ passes the Russian Doll test because we can nest $G(\tau_{candidate})$ in the slack of $G(\tau_2)$. Likewise, in Table 2, we can order the groups according to t_{max} and find that t_δ for a given group is greater than or equal to t_{max} of a group with a smaller t_{max}.

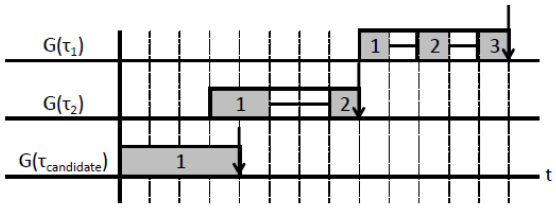

Figure 12. Visual description of a Russian Dolls Tests for a candidate task $\tau_{candidate}$, where $\tau_{candidate}$ fails the test. We can see that the window of opportunity, t_{max}, for $G(\tau_{candidate})$ is larger than the slack, t_δ, allotted by $G(\tau_2)$. Parameters of the test are shown in Table 3.

Task Group	t_{max}	t_{min}	t_δ
$G(\tau_1)$	13	5	8
$G(\tau_2)$	8	5	3
$G(\tau_{candidate})$	4	4	0

Table 3. Table of values for an example of a Russian Dolls Test corresponding to the right example in Figure 12.

In Figure 12, we show a situation where $\tau_{candidate}$ fails the Russian Doll Test. $\tau_{candidate}$ fails the test because we cannot nest $G(\tau_{candidate})$ within the slack of $G(\tau_2)$. Likewise, in Table 3, if we order the groups according to t_{max}, t_δ for group $G(\tau_2)$ is not larger than t_{max} of group $G(\tau_{candidate})$ with a smaller t_{max}.

IV.A.1. Russian Dolls Test - Nuts and Bolts

For the case of a uniprocessor (single-agent) Russian Dolls Test, we can formulate t_{max}, t_{min}, and t_δ for a given task, $\tau_{i,j}$, as shown in Equations 8-10.

$$t_{max} = UB_{APSP}\left(G_{first,unexecuted}(\tau_{i,j}), G_{last,unexecuted}(\tau_{i,j})\right) \tag{8}$$

where $UB_{APSP}\left(G_{first,unexecuted}(\tau_{i,j}), G_{last,unexecuted}(\tau_{i,j})\right)$ returns the upper bound of the APSP between the start of the first, unexecuted task in $\tau_{i,j}$'s task group and the end of the last, unexecuted task in $\tau_{i,j}$'s group.

$$t_{min} = LB_{APSP}\left(G_{first,unexecuted}(\tau_{i,j}), G_{last,unexecuted}(\tau_{i,j})\right) \tag{9}$$

where $LB_{APSP}\left(G_{first,unexecuted}(\tau_{i,j}), G_{last,unexecuted}(\tau_{i,j})\right)$ returns the lowerbound of the APSP between the start of the first, unexecuted task in $\tau_{i,j}$'s task group and the end of the last, unexecuted task in $\tau_{i,j}$'s group.

$$t_\delta = t_{max} - t_{min} \tag{10}$$

Recall the three conditions of the online consistency check that govern CR-EDF, described in Section IV. The third condition states that executing a complex deadline constraint is mutually exclusive with executing another deadline constraint. This condition is important to ensure the validity of the Russian Dolls Test.

Let us consider a set of nested, Russian Dolls. If one of the dolls suddenly were to suddenly shrink in size, then it would crush the smaller dolls nested inside. Similarly, when considering the scheduling problem of nesting task groups, if the slack of a group shrinks relative to a smaller task group (i.e., a smaller t_{max}), then we could not guarantee that there was enough time to execute the smaller task groups.

More formally, if we schedule a deadline task group, $G(\tau_{i,j})$ where $\tau_{i,j}$ is the free task which begins the associated deadline constraint, $d_{(i,j),(x,y)}$; then, we must insure the property shown in Equation 11.

$$t_\delta|_{t=s_{i,j}} \leq t_\delta|_{t=t_n} \forall s_{i,j} \leq t_n < f_{x,y} \tag{11}$$

where $\tau_{x,y} = G_{end}(\tau_{i,j})$. In the case that $d_{(i,j),(x,y)}$ is a task-task deadline constraint, then $x = i$ and $y \geq j$. In the case of an epoch-task deadline constraint, $x = 1$ and $y > j$.

We will now consider an example deadline task group, $G(\tau_{2,2} = \{\tau_{2,2}, \tau_{2,3}, \tau_{2,4}\}$, with a complex deadline, from the well-formed task model shown in Figure 6 to illustrate such a situation where a deadline task group's slack, t_δ, might shrink. We first evaluate $\tau_{candidate} = \tau_{2,2}$, and determine that

$$\begin{aligned} t_{max}|_{t=s_{\tau_{2,2}}} &= UB_{APSP}\left(G_{first,unexecuted}(\tau_{2,2}), G_{last,unexecuted}(\tau_{2,2})\right) \\ &= UB_{APSP}\left(\tau_{2,2}, \tau_{2,4}\right) \\ &= d_{\{(2,2),(2,4)\}} \end{aligned} \tag{12}$$

$$\begin{aligned} t_{min}|_{t=s_{\tau_{2,2}}} &= LB_{APSP}\left(G_{first,unexecuted}(\tau_{2,2}), G_{last,unexecuted}(\tau_{2,2})\right) \\ &= LB_{APSP}\left(\tau_{2,2}, \tau_{2,4}\right) \end{aligned} \tag{13}$$

$$\begin{aligned} t_{slack}|_{t=s_{\tau_{2,2}}} &= t_{max}|_{t=s_{\tau_{2,2}}} - t_{min}|_{t=s_{\tau_{2,2}}} \\ &= d_{\{(2,2),(2,4)\}} - LB_{APSP}\left(\tau_{2,2}, \tau_{2,4}\right) \end{aligned} \tag{14}$$

Second, we determine the slack for the task group after we have executed $\tau_{2,2}$ and $\tau_{candidate} = \tau_{2,3}$. For now, we will assume that $d_{\{(2,3),(2,4)\}}$ is non-trivial (i.e., $d_{\{(2,3),(2,4)\}}$ yields a tighter upperbound between $\tau_{2,3}$ and $\tau_{2,4}$ than if those tasks were not constrained by the $d_{\{(2,3),(2,4)\}}$).

$$t_{max}|_{t=s_{\tau_{2,3}}} = d_{\{(2,3),(2,4)\}} \tag{15}$$

$$t_{min}|_{t=s_{\tau_{2,3}}} = LB_{APSP}\left(\tau_{2,3}, \tau_{2,4}\right) \tag{16}$$

$$t_{slack}|_{t=s_{\tau_{2,3}}} = d_{\{(2,3),(2,4)\}} - LB_{APSP}\left(\tau_{2,3}, \tau_{2,4}\right) \tag{17}$$

Recall that, For the Russian Dolls method to be valid, we must ensure that the slack associated with a task group does not decrease with time. If this is true, then

$$\begin{aligned} t_{slack}|_{t=s_{\tau_{2,2}}} &\leq t_{slack}|_{t=s_{\tau_{2,3}}} \\ d_{\{(2,2),(2,4)\}} - LB_{APSP}\left(\tau_{2,2}, \tau_{2,4}\right) &\leq d_{\{(2,3),(2,4)\}} - LB_{APSP}\left(\tau_{2,3}, \tau_{2,4}\right) \\ d_{\{(2,2),(2,4)\}} - d_{\{(2,3),(2,4)\}} &\leq LB_{APSP}\left(\tau_{2,2}, \tau_{2,4}\right) - LB_{APSP}\left(\tau_{2,3}, \tau_{2,4}\right) \\ d_{\{(2,2),(2,4)\}} - d_{\{(2,3),(2,4)\}} &\leq c_{2,2} + \theta_{2,2} \\ d_{\{(2,2),(2,4)\}} &\leq c_{2,2} + \theta_{2,2} + d_{\{(2,3),(2,4)\}} \end{aligned} \tag{18}$$

When Equation 18 does not hold, then the Russian Dolls Test is invalid because we cannot ensure that nesting a candidate task within $G(\tau_{2,2})$ will yield sufficient slack for the candidate's task group to finish executing within the specified deadlines. For example, consider the situation where an outer deadline, such as $d_{\{(2,2),(2,4)\}}$ constrains a task group with complex deadlines, yielding more slack than is allowed for the inner deadline, such as $d_{\{(2,3),(2,4)\}}$.

To mitigate the risk of such occurrences, CR-EDF allows multiple deadline constraints to be active at the same time if and only if none of those constraints are complex deadline constraints. This restriction is enforced by Condition 3 described above.

Recall our assumption that $d_{\{(2,3),(2,4)\}}$ was nontrivial; for Equation 18 to hold, $d_{\{(2,3),(2,4)\}}$ would have to be a trivial deadline. In this situation, we could prune $d_{\{(2,2),(2,4)\}}$ from the set of deadline constraints, $d_{\{(2,2),(2,4)\}}$ would no longer be a complex deadline constraint, and there would not be any complex tasks in the network. In general, this pruning is advantageous because it would allow for more deadlines to be active concurrently, thus reducing processor idle time. We will address techniques for pruning deadline constraints to reduce the makespan in future work.

IV.B. Putting it all Together

We recall that CR-EDF is a scheduling policy that maintains a queue of all active tasks and attempts to schedule tasks that have been released, according to deadline priority, by performing a three-fold, online consistency check. We will now walk through pseudo-code describing the CR-EDF scheduling policy.

CREDFScheduler()

```
1: for (t = 0 → t_n) do
2:     if processorIsIdle then
3:         availableTasks = getAvailableTasks(); //According to Condition 1
4:         for (i = 1 → card(availableTasks)) do
5:             τ_candidate = availableTasks(i);
6:             if ({τ_active} = ∅) then
7:                 schedule(τ_candidate);
8:             else
9:                 if (τ_active ∉ {τ_CDC}) then //In accordance with Condition 3
10:                    if russianDollsTest(τ_candidate); //According to Condition 2 then
11:                        schedule(τ_candidate);
12:                    end if
13:                end if
14:            end if
15:            if isScheduled(τ_candidate) then
16:                break();
17:            end if
18:        end for
19:    end if
20: end for
```

Line 1 instantiates a method to step through time, and Line 2 checks whether or not the processor is currently idle. If the processor has finished its previous task, then, in lines 3-4, we attempt to schedule a new task. Line 5 gets the next available task with the i^{th} earliest deadline. Tasks are available only if those tasks have been released (Definition 2) and satisfy Condition 1 from Section IV. Lines 6-7 schedule the candidate task if there are no active tasks (Definition 7).

In Line 8, if there are active tasks, and, in Line 9, if the candidate task is not a complex deadline constraint as per Condition 3, then, in Line 10, we perform the Russian Dolls Test as per Condition 2 (Section IV.A). If $\tau_{candidate}$ passes the Russian Dolls Test, then we schedule $\tau_{candidate}$. In Lines 15-16, if we were able to schedule the candidate task, then we go to the next time step. Similarly, if we were unable to schedule any tasks at this time step or the processor is currently executing a task, we step through to the next time step.

V. Results

In this section, we empirically validate that the scheduler is near-optimal for real-world task scenarios. To do so, we must first define the theoretical lower and upperbound makespans for the scheduler.

V.A. Benchmarks

The makespan returned by the scheduler, m_{SCH}, is bounded by theoretical upper and lowerbound constraints, which are defined in Equations 19 and 20. We note that this theoretical lowerbound is not necessarily feasible, and is less than or equal to the optimal solution for the network. Because calculating the optimal m for networks with hundreds of tasks is too computationally expensive, we rely on the theoretical lowerbound defined in Equation 19 as a benchmark.

$$m_{LB} = \left(\sum_{i=1}^{nrows} \sum_{j=1}^{ncols} c_{i,j} \right) \tag{19}$$

We can also construct an upperbound for the makespan, m_{UB}.

$$m_{UB} = m_{LB} + \left(\max_{i}(\theta_{i,1}) + \sum_{i}^{nrows} \sum_{j=2}^{ncols-2} \theta_{i,j} + \max_{i}(\theta_{i,ncols-1}) \right) \tag{20}$$

For tasks combined using the *serial task combinator*, wait constraints joining the tasks contribute their full duration to the total execution time in the worst case. For parallel components, however, the largest wait is contributed by the maximum wait constraint in each joining column (i.e., the first and last columns in the network). Thus, to find m_{UB}, we take the sum over all of the wait constraints that combine tasks through serial precedence, and we take the maximum over the rows for the wait constraints that join tasks in parallel.

For our empirical validation, we will use m_{LB} as our benchmark and will measure the percent difference between m_{SCH} and m_{LB}. For context, we will also show the percent difference between m_{UB} and m_{LB}.

V.B. Empirical Validation

Empirical results are produced using a random generator for well-formed task sets. The generator takes as input the number of tasks ranging $[100, 400]$ (not including the epoch and terminus tasks), fraction of deadline constrained tasks, $\hat{d}$, ranging $[0, 0.7]$, and fraction of non-zero wait constraint, $\hat{\theta}$, ranging $[0.25, 1.0]$. Task costs and wait durations are generated from a normal distribution with $\mu = 4.5$ and $\sigma = 4.5$. Our CR-EDF scheduling policy are implemented in MATLAB, and all computation was performed on a Dell XPS, 3.3 GHz, Quadcore computer.

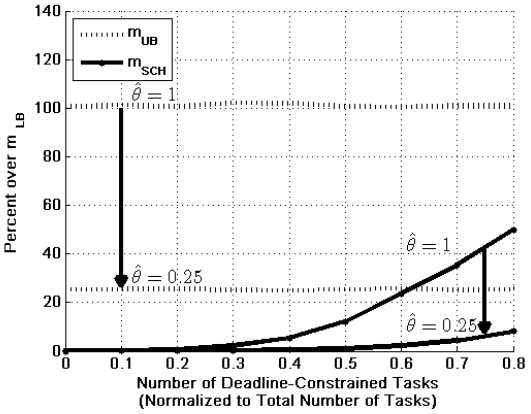

Figure 13. Percent difference between m_{WC}, m_{SCH} and the sum of the task costs for $\hat{d}$ ranging in [0,0.9] and $\hat{\theta} = \{0.25, 1.0\}$; $nTasks = 400$.

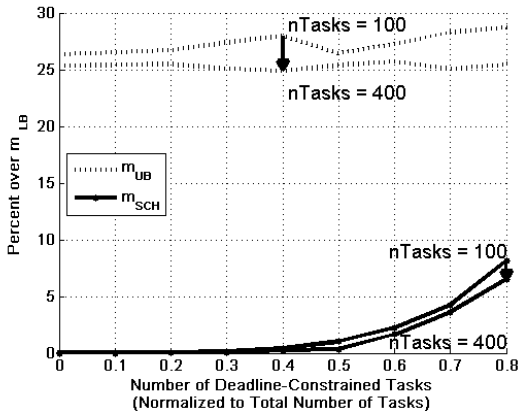

Figure 14. Percent difference between m_{WC}, m_{SCH} and the sum of the task costs for $\hat{d}$ ranging in [0,0.9] and $nTasks = \{100, 400\}$; $\hat{\theta} = 0.25$.

VI. Discussion

The performance of the CR-EDF scheduling policy is a function of various parameters defining the well-formed task model. First, we will consider the effect of the number of nonzero wait constraints in the task

network as a proportion of the total number of tasks, $\hat{\theta}$, versus the number of deadline-constrained tasks in the network as a proportion of the total number of tasks, $\hat{d}$. Second, we will evaluate the effect of increasing the total number of tasks in the network as a function of $\hat{d}$.

In Figure 13, we can see the effects of $\hat{\theta}$ and $\hat{d}$. The makespan returned by the scheduler, m_{SCH}, is close the theoretical lowerbound for $\hat{\theta} = [0.25, 1.00]$ and for $\hat{d}$ ranging $[0, 0.4]$. While m_{SCH} does deviate from the theoretical lowerbound for $\hat{\theta} = 1.00$, $\hat{d} \geq 0.4$, we recall that this theoretical lowerbound is merely the sum of the task costs and does not represent the optimal schedule.

In practice, real-world task sets for flexible scheduling of factory robots or tasking of unmanned vehicles for long-duration missions involve $nTasks \geq 100$ and $\hat{\theta} \leq 0.25$. As such, we will extend our investigation to determining the performance of m_{SCH} as a function of the total number of tasks versus $\hat{d}$ where $\hat{\theta} = 0.25$. In Figure 14, we see that the size of the network does not have a significant effect on the performance of m_{SCH} relative to the theoretical lowerbound. For both cases, where the number of tasks is varied from 100 to 400, m_{SCH} remains within a few percent of the lowerbound. For the most constrained networks, m_{SCH} is approximately 8% over m_{LB}; however, we find in practice that real-world task networks are significantly more flexible ($\hat{d}$ ranging $[0, .5]$) where the corresponding percent over the m_{LB} is approximately $0\% - 2\%$.

VII. Conclusion

We have presented an idling, dynamic priority scheduling policy for non-preemptive task sets with precedence, wait constraints, and deadline constraints. Our polynomial-time scheduling policy operates on a well-formed task model where tasks are related through a hierarchical temporal constraint structure found in many real-world applications. Although the CR-EDF policy is not optimal, we show through empirical evaluation that the policy produces schedules that are within a few percent of the best possible makespan.

VIII. Future Work

In future work, we will develop a polynomial-time schedulability test for the CR-EDF scheduling policy, and extend both to the case of multiple agents. Furthermore, we will apply our well-formed task model and the CR-EDF scheduler and schedulability test to various real-world applications outlined in Section II.E. We will also extend the work to process nested, well-formed task models and non-deterministic task costs and wait constraints.

Acknowledgment

Funding for this project was provided in part by Boeing Research and Technology and in part by the National Science Foundation (NSF) Graduate Research Fellowship Program (GRFP) under grant number 2388357.

References

[1]A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: an open platform for satisfiability modulo theories. In *Proc. 2nd workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[2]G. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2005.

[3]J. Cates. Route optimization under uncertainty for unmanned underwater vehicles. Master's thesis, 2011.

[4]H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990.

[5]L. Cucu-Grosjean and Yves Sorel. Schedulability conditions for systems with precedence and periodicity constraints without preemption. In *Proc. International Conference on Real-Time Systems*, April 2003.

[6]Liliana Cucu-Grosjean, N. Pernet, and Yves Sorel. Periodic real-time scheduling: from deadline-based model to latency-based model. *Annals of Operations Research*, 159:41–51, 2008.

[7]Liliana Cucu-Grosjean and Yves Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proc. International Conference on Real-Time Systems*, RTS'02, April 2002.

[8]Rina Dechter, Italy Meiri, and Judea Pearl. Temporal constraint networks. *AI*, 49(1), 1991.

[9]M. Garey, D. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2), 1976.

[10]Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[11]Michael R. Garey, David S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, May 1976.

[12]Laurent George, Paul Muhlethaler, and Nicolas Rivierre. Optimality and non-preemptive real-time scheduling revisited. In *Proc. INRIA*, April 1995.

[13]D. Bohme I. Anagnostakis, J.-P. Clarke and U. Volckers. Runway operations planning and control: Sequences and scheduling. In *Proc. International Conference on System Sciences, 2001*, 2001.

[14]U. Ozguner L. Xu. Battle management for unmanned aerial vehicles. In *Proc. IEEE CDC*, 2003.

[15]B. Nelson and T. K. Kumar. CircuitTSAT: A solver for large instances of the disjunctive temporal problem. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors, *Proc. ICAPS*, 2008.

[16]M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, second edition, 2006.

[17]David E. Smith, Feremy Frank, and Ari Jónsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15, 2000.

[18]H. Zhao, L. George, and S. Midonnet. Worst case response time analysis of sporadic task graphs with edf non-preemptive scheduling on a uniprocessor. In *Proc. ICAS*, ICAS '07, Washington, DC, USA, 2007. IEEE Computer Society.