The Digraph Real-Time Task Model

Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi
Uppsala University, Sweden
Email: {martin.stigge | pontus.ekberg | nan.guan | yi}@it.uu.se

***Abstract*—Models for real-time systems have to balance the inherently contradicting goals of expressiveness and analysis efficiency. Current task models with tractable feasibility tests have limited expressiveness, restricting their ability to model many systems accurately. In particular, they are all *recurrent*, preventing the modeling of structures like mode switches, local loops, etc.**

In this paper, we advance the state-of-the-art with a model that is free from these constraints. Our proposed task model is based on arbitrary directed graphs (digraphs) for job releases. We show that the feasibility problem on preemptive uniprocessors for our model remains tractable. This even holds in the case of task systems with arbitrary deadlines.

I. Introduction

In the design of real-time systems, abstract models are used to validate non-functional properties like timing behavior in schedulability analysis. Designers face the problem of choosing an appropriate level of abstraction in order to satisfy two contradicting goals. First, the model should be sufficiently *expressive* to enable modeling of the system's behavior as precisely as possible. This contributes to easier modeling without restrictions and to reducing wasteful resource over-provisioning. Second, the analysis of the model should be *efficient* in order to scale well with the system's size and provide results within a reasonable time frame.

The well-known *Liu and Layland task model* [1] which characterizes task behavior by relatively few parameters (execution time and period of task activations) has been thoroughly studied since the 1970s. It does very well on the analysis efficiency side but it is very restrictive on the type of tasks allowed. At the other extreme, task activations may be modeled using formalisms as powerful as *timed automata* [2]. This allows very accurate system models, but at the price of very costly or even impossible schedulability tests.

With analysis efficiency as a major concern, researchers have proposed increasingly expressive models over the years (see Figure 1), for all of which feasibility analysis can be done efficiently. The most general model is the *non-cyclic recurring real-time task model* proposed in [3]. It models each task using a directed acyclic graph (DAG). Each vertex models the release of a new job. Different vertices represent different types of jobs with potentially different execution time and deadline parameters. Further, job inter-release separation times are provided as labels on the graph edges. Recurrent behavior is modeled by adding back edges from sink vertices to the unique source vertex, also labeled with individual inter-release separations. Although feasibility analysis for this model has been shown to be tractable [3], critical restrictions for system modeling still remain. For example, unbounded local looping or task mode switches cannot be expressed, and models in general are inflexible.

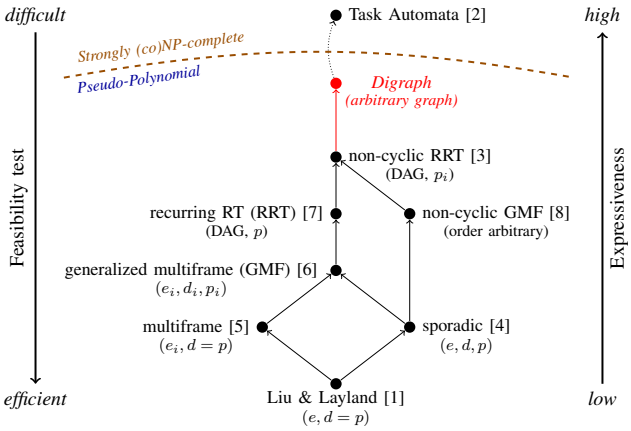

Fig. 1. A hierarchy of task models. Arrows indicate the generalization relationship. The higher in the hierarchy, the higher the expressiveness, but also the more expensive the feasibility test.

To enable these features, we propose a model that allows arbitrary directed graphs to represent the release structure of jobs in terms of order and timing. Our *digraph real-time task model* (DRT) is a significant generalization of previous models, providing a major increase in expressiveness. This makes DRT the most general tractable model currently known. We show how the feasibility analysis problem can be solved in pseudo-polynomial time, even though it becomes considerably more difficult, since an unbounded number of paths through the task graph needs to be handled. In particular, this paper provides the following contributions:

- We introduce a *path abstraction technique* to prevent search space explosion via dynamic programming.
- We show that the feasibility problem in our proposed model remains *tractable*, i.e., can be decided in pseudo-polynomial time for systems with bounded utilization.
- We study the extension of our model to *arbitrary deadlines* and show that feasibility remains tractable. This has not been shown before, not even for many of the previous, less expressive models.

A prototype implementation of our technique shows that it is applicable to large task sets. We are able to analyze randomly generated task sets of 100 tasks with 20 job types each and both low and high system utilizations within a few seconds.

A. Prior Work

Early works on generalizing the Liu & Layland task model include the *sporadic* [4] and the *multiframe* [5] task models, which were later unified in the *generalized multiframe (GMF)* model [6]. In summary, GMF decouples task periods from deadlines, allows *sporadic* job releases and provides a set of different job types ("frames") through which the task cycles. Despite these generalizations, feasibility analysis has been shown to be tractable by proposing tests that run in pseudo-polynomial time. The model was further generalized in the *recurring real-time (RRT)* task model [7] by relaxing the "linear" order in which different job types are released by a task. RRT allows *branching code* to be modeled by a DAG, thereby greatly improving modeling expressiveness. The *non-cyclic GMF* proposed in [8], generalizes GMF in another direction. Here, jobs may be released in any order, therefore the behavior is not necessarily cyclic. An attempt to unify both divergent generalizations has been made in recent work [3] with the introduction of the *non-cyclic RRT* model. It adds non-cyclic behavior to RRT by allowing "restarts" of the DAG traversal to be dependent on the last job released. Using a dynamic programming method from [9], it is shown in [3] that feasibility can be decided in pseudo-polynomial time also for the non-cyclic RRT model.

At the other end of the expressiveness spectrum, the *task automata* model [2] allows many features such as complex dependencies between job release times and task synchronization. However, schedulability analysis is very expensive and is even shown to be undecidable in certain variants of the model.

A related model for schedulability analysis is the *Real-Time Calculus* (RTC) from [10]. It enables compositional analysis of timed systems using the *arrival curve* model and is similar to the demand bound function that we use in our feasibility analysis. However, our digraph task model allows more precise expression of timing constraints between job releases, which have to be abstracted in the RTC.

II. The Digraph Real-Time Task Model

A digraph real-time (DRT) task system $\tau = \{T_1, \ldots, T_N\}$ consists of N independent tasks. A task T is characterized by a *directed graph* $G(T)$. The set $\{v_1, \ldots, v_n\}$ of vertices of $G(T)$ represents the types of jobs that can be released by that task. Each vertex v_i is labeled with an ordered pair $\langle e(v_i), d(v_i)\rangle$ where $e(v_i)$ is the worst-case execution-time demand of the corresponding job, and $d(v_i)$ its relative deadline. Both are assumed to be positive integers. The edges of the graph represent the order in which jobs generated by T are released. Each edge (u, v) is labeled with a positive integer $p(u, v)$ for the minimum job inter-release separation time.

Note that this model is a strict generalization of the non-cyclic RRT model of [3]. We allow arbitrary graphs for task modeling. In particular, we allow *arbitrary cycles* in $G(T)$ and do not have distinct source and sink vertices. We do not require a special initial vertex nor repeated revisits of a particular vertex.

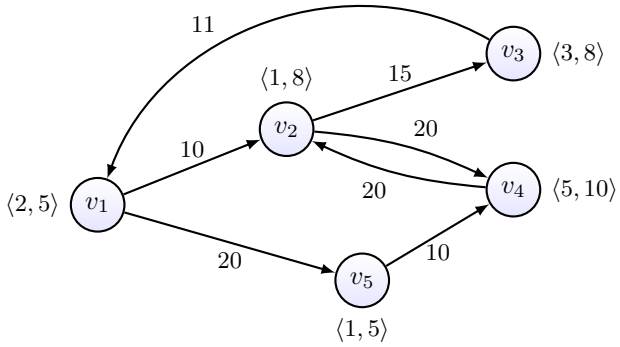

Fig. 2. An example task containing five different types of jobs

In the first part of this paper, we assume that the task systems satisfy the *Frame Separation property*[1] by which all jobs' deadlines are constrained to not exceed the inter-release separation times: for all vertices u and their outgoing edges (u, v) we require $d(u) \leqslant p(u, v)$. We remove this constraint in Section VI and show that the results still hold.

Example II.1. *Figure 2 shows an example of a DRT task satisfying the Frame Separation property. We will use it as a running example throughout the rest of the paper.*

Semantics: An execution of task T corresponds to a potentially infinite path in $G(T)$. Each visit to a vertex along that path triggers the release of a job with parameters specified by the vertex labels. The job releases are constrained by inter-release separation times specified by the edge labels. Formally, we use a 3-tuple (r, e, d) to denote a *job* that is released at (absolute) time r, with worst-case execution time e and deadline at (absolute) time d. We assume dense time, i.e., $r, d \in \mathbb{R}_{\geqslant 0}$. A job sequence $\sigma = [(r_1, e_1, d_1), (r_2, e_2, d_2), \ldots]$ is *generated by* T, if and only if there is a (potentially infinite) path $\pi = (\pi_1, \pi_2, \ldots)$ in $G(T)$ satisfying for all i:

1) $e_i = e(\pi_i)$,
2) $d_i = r_i + d(\pi_i)$,
3) $r_{i+1} - r_i \geqslant p(\pi_i, \pi_{i+1})$.

For a task set τ, a job sequence σ is *generated by* τ, if it is a composition of sequences $\{\sigma_T\}_{T\in\tau}$, which are individually generated by the tasks T of τ.

Example II.2. *For the example task T in Figure 2, consider the job sequence $\sigma = [(5, 5, 15), (25, 1, 33), (42, 3, 50)]$. It corresponds to the path $\pi = (v_4, v_2, v_3)$ in $G(T)$ and is thus generated by T. Note that, while the second job in σ (associated with v_2) is released as early as possible after the first job (v_4), the same is not true for the third job (v_3). This "sporadic" behavior is valid in the semantics of our task model.*

[1] Strictly speaking, the *l-MAD* property from [6] which is a bit less restrictive would be sufficient. However, for simplicity of presentation, we assume $d(u) \leqslant p(u, v)$ for Sections III to V and remove all constraints in Section VI.

III. Feasibility Analysis

For our proposed task model, we are interested in solving the associated feasibility problem:

Definition III.1 (Feasibility)**.** *A task set τ is* preemptive uniprocessor feasible*, if and only if all job sequences generated by τ can be executed on a preemptive uniprocessor platform such that all jobs meet their deadlines.*

In particular, for a job (r, e, d) to be scheduled successfully, there must be an accumulated duration of e time units where the job executes exclusively on the processor within the time interval $[r, d]$. It is known that Earliest Deadline First (EDF) is an optimal scheduling algorithm for scheduling independent jobs on a preemptive uniprocessor. Thus, the *feasibility* problem is equivalent to *EDF schedulability*.

In order to show an efficient decision procedure for the DRT task model, we first introduce the general framework (Section III-A). After that, we focus on the two main problems to be solved during the analysis (Sections IV and V).

A. The Demand Bound Function

Our analysis uses the concept of a *demand bound function (dbf)*, which is an established framework in schedulability theory. Intuitively, a *dbf* expresses the accumulated execution time that a task set can *demand* from the processor within any time interval of given length. In particular, it considers each execution requirement that is both *released* within the interval and needs to be *finished* before the end of the interval. Formally:

Definition III.2 (Demand Bound Function)**.** *For a task T and an interval length t, $dbf_T(t)$ denotes the* maximum *cumulative execution requirement of jobs with both release time and deadline within an interval of length t, over all job sequences generated by T. Further, for a task set τ,*

$$dbf(t) := \sum_{T \in \tau} dbf_T(t).$$

By definition, *dbf* is *tight* in the sense that for each t, there is a job sequence generated by task set τ in which some jobs actually have an execution demand of $dbf(t)$ within an interval of t time units. Note that the definition of $dbf(t)$ as the sum of $dbf_T(t)$ of all tasks T relies on their independence of each other. Note further that we assume time to be dense, so *dbf* is defined for all $t \in \mathbb{R}_{\geqslant 0}$. However, changes clearly only occur at integers.

Example III.3. *Consider again job sequence $\sigma = [(5, 5, 15), (25, 1, 33), (42, 3, 50)]$ from Example II.2, generated by task T from Figure 2. This sequence σ shows that in a time interval $t = 45$, task T may generate a demand of 9 on the processor as follows: The first job is released at $t_1 = 5$ and the third job has its deadline at $d_3 = 50$. Thus, all three jobs of the sequence have both their release time and deadline within time interval $[5, 50]$ of length 45. Together, their execution time is $5 + 1 + 3 = 9$.*

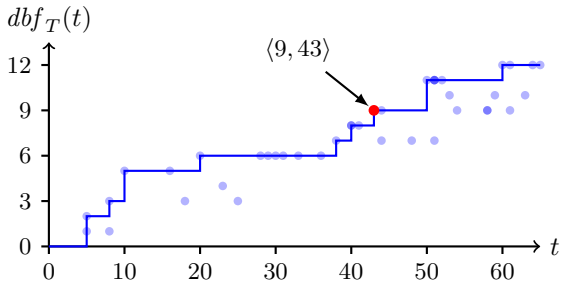

Fig. 3. Demand bound function for the DRT task in Figure 2. The dots depict the demand pairs of that task. Demand pairs corresponding to multiple paths have a darker color.

In fact, there are no job sequences generated by T with a higher demand within an interval of length 45. We can conclude that $dbf_T(45) = 9$ for our example task T.

A tight demand bound function can be used in a feasibility test, thanks to the following proposition.

Proposition III.4. *A DRT task system τ is preemptive uniprocessor feasible if and only if:*

$$\forall t \geqslant 0 : dbf(t) \leqslant t$$

A proof of this can be established in a very similar way as in previous work, e.g. [6], and we omit it here. Intuitively, there must be a feasible schedule for all job sequences generated by τ if and only if, for all time interval lengths t, the execution time demand of τ fits into that interval.

As a consequence, the feasibility of a DRT task system can be checked by determining whether there exists a t_f violating this property, i.e., such that $dbf(t_f) > t_f$. In order to check this, we need to solve two main problems:

1) How do we compute $dbf(t)$ for a given t?
2) For which t do we need to compute $dbf(t)$?

The second problem is important since if we were to naïvely check $dbf(t)$ sequentially for all integers t, we would not terminate when analyzing feasible task systems. In Section V we will therefore derive a pseudo-polynomial bound for t_f.

B. Demand Pairs

The first problem of how to compute $dbf(t)$ reduces to calculating $dbf_T(t)$ for each task T. In Figure 3, $dbf_T(t)$ for the task in Figure 2 is (partially) shown. Note that it only changes at certain points ("steps"). Each point corresponds to a path π in $G(T)$. Take for example the marked point $t = 43$ with $dbf_T(43) = 9$. It corresponds to path $\pi = (v_4, v_2, v_3)$ from Example II.2: π has an execution requirement of 9 time units as shown in Example III.3. Further, if all jobs are released as early as possible , the time interval between the release of the first job and the deadline of the third job is 43 time units. The pair $\langle 9, 43 \rangle$ of execution time and time interval gives useful information in the process of calculating the demand bound function. It can be interpreted as a generalization of execution time demand and relative deadline from a single job to a job sequence, as follows.

Definition III.5 (Demand Pair)**.** *For a finite path* $\pi = (\pi_0, \ldots, \pi_l)$ *in* $G(T)$ *we define:*

$$\text{Execution demand:} \quad e(\pi) := \sum_{i=0}^{l} e(\pi_i)$$

$$\text{Deadline:} \quad d(\pi) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + d(\pi_l)$$

We call $\langle e(\pi), d(\pi) \rangle$ *a* demand pair *that corresponds to* π.

For each path π in $G(T)$ of our running example, Figure 3 contains a dot for the demand pair $\langle e(\pi), d(\pi) \rangle$. If we were able to compute all demand pairs, we could easily compute the demand bound function, since

$$dbf_T(t) = \max \{ e \mid \langle e, d \rangle \text{ demand pair with } d \leqslant t \}.$$

As a motivation of this formula, recall Definition III.2 of the demand bound function. Given a t and the demand pairs $\langle e, d \rangle$ of all paths, the value of $dbf(t)$ is the maximal execution demand e of all paths with a deadline d of *at most* t. In particular, this also includes paths with much shorter deadline.

A naïve approach could be to enumerate all paths and compute the corresponding demand pairs. However, this would be an exponential procedure. Next, we show how to compute all relevant demand pairs within a pseudo-polynomial time bound.

IV. Calculating Demand Pairs

A key observation for calculating demand pairs efficiently is that several different paths may correspond to the same demand pair.

Example IV.1. *In our running example from Figure 2, consider the following three paths:*

$$\pi^A = (v_3, v_1, v_5, v_4)$$
$$\pi^B = (v_3, v_1, v_2, v_4)$$
$$\pi^C = (v_4, v_2, v_3, v_1)$$

All three paths correspond to the demand pair $\langle 11, 51 \rangle$.

Using demand pairs as an *abstraction* of concrete paths through the task graph reduces the number of objects that need to be tracked. In order to develop this into an efficient iterative procedure, we need to extend the abstraction. Note that in the above example, paths π^A and π^B (in contrast to π^C) not only correspond to the same demand pair, but also *end* in the same vertex of $G(T)$. Any extension of both paths will therefore always result in the same demand pairs for both extended paths. In an iterative procedure, only one (or rather, an abstraction of both) of them needs to be kept. This is the key to preventing an exponential explosion in complexity.

We formalize these observations as follows.

Definition IV.2 (Demand Triple)**.** *For a finite path* $\pi = (\pi_0, \ldots, \pi_l)$ *in* $G(T)$*, we call* $\langle e(\pi), d(\pi), \pi_l \rangle$ *a* demand triple. *We say that* π corresponds *to that demand triple.*

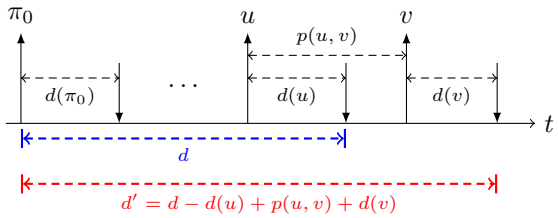

Fig. 4. Extending a demand triple $\langle e, d, u \rangle$ by a vertex v. Let $\langle e, d, u \rangle$ correspond to a path $\pi = (\pi_0, \ldots, \pi_{l-1}, u)$ and $\langle e', d', v \rangle$ correspond to its extension $\pi' = (\pi_0, \ldots, \pi_{l-1}, u, v)$. Depicted is a job release sequence generated by π' in which all jobs are released as early as possible. The relation between d and d' is shown.

Lemma IV.3. *For each constant* $D \in \mathbb{N}$*, the number of demand triples* $\langle e, d, v \rangle$ *with* $d \leqslant D$ *is bounded polynomially in* D *and* n*, the number of vertices in* $G(T)$.

Proof: We may assume $e \leqslant d$ since otherwise there is clearly a deadline violation. Thus, all demand triples are in $\mathbb{N}_{\leqslant D} \times \mathbb{N}_{\leqslant D} \times \mathbb{N}_{\leqslant n}$, leaving only $\mathcal{O}(D^2 n)$ possibilities. ∎

Using these insights, one can implement an iterative procedure for calculating $dbf_T(t)$ for a given t using a graph traversal that stores only demand triples. We first give a high-level description:

1) Consider all paths of length 0, i.e., all vertices of $G(T)$, and store their corresponding demand triples.
2) For a stored demand triple $\langle e, d, u \rangle$, consider all successor vertices v of u. For each such v, one can compute a new demand triple $\langle e', d', v \rangle$ corresponding to a path that has been extended by v. In particular,

   $$e' = e + e(v) \text{ and } d' = d - d(u) + p(u, v) + d(v).$$

   If π is a path corresponding to $\langle e, d, u \rangle$, then this computation ensures that $\langle e', d', v \rangle$ corresponds to π extended by vertex v. See Figure 4.
3) Each newly computed demand triple $\langle e', d', v \rangle$ is stored only if:
   - It is not stored yet, and
   - $d' \leqslant t$
4) Repeat until there are no new demand triples.
5) Using all demand triples $\langle e, d, v \rangle$, calculate $dbf_T(t)$:

   $$dbf_T(t) = \max \{ e \mid \langle e, d, v \rangle \text{ demand triple with } d \leqslant t \}.$$

Note that the above procedure only needs to discover paths π (or rather, the demand triple abstractions thereof) which have a path length of at most t, since with each new vertex, $d(\pi)$ increases by at least one. We give a formal, more detailed algorithm in Figure 5 which is based on this observation. For simplicity of presentation, it discovers all paths in a breadth-first manner. In particular, the sets DT_k contain all demand triples corresponding to paths of length k.

1: $DT_0 \leftarrow \{\langle e(v_i), d(v_i), v_i\rangle \mid v_i \text{ vertex of } G(T)\}$
2: **for** $k = 1$ **to** t **do**
3: $\quad DT_k \leftarrow \emptyset$
4: $\quad$ **for all** $\langle e, d, u\rangle \in DT_{k-1}$ **do**
5: $\quad\quad$ **for all** edges (u, v) in $G(T)$ **do**
6: $\quad\quad\quad e' \leftarrow e + e(v)$
7: $\quad\quad\quad d' \leftarrow d - d(u) + p(u, v) + d(v)$
8: $\quad\quad\quad$ **if** $d' \leqslant t$ **then**
9: $\quad\quad\quad\quad DT_k \leftarrow DT_k \cup \{\langle e', d', v\rangle\}$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end for**
14: **return** $\max\left\{e \mid \langle e, d, v\rangle \in \bigcup_{k\leqslant t} DT_k\right\}$

Fig. 5. An iterative algorithm for calculating $dbf_T(t)$.

Correctness: By the intuitive reasoning above, the algorithm in Figure 5 correctly tracks all paths through $G(T)$. It only discards paths if their deadline already is too long, or if it discovers a path corresponding to an already considered demand triple. The full proof of the correctness lemma is given in Appendix A.

Lemma IV.4. *The algorithm in Figure 5 is correct. In particular:*

1) *There is a path π in $G(T)$ with $d(\pi) \leqslant t$ for which $e(\pi)$ is equal to the return value.*
2) *There is no path π' in $G(T)$ with $d(\pi') \leqslant t$ for which $e(\pi')$ strictly exceeds the return value.*

Complexity: The demand triple abstraction makes sure that for each k, there can be at most polynomially many demand triples in DT_k (Lemma IV.3). This bounds the number of iterations of line 4. Further, for each demand triple, there are at most n extensions (maximum number of successors), which also bounds number of iterations of line 5. Finally, finding the maximum in line 14 can be done in time linear in the set size. We summarize the result in the following lemma.

Lemma IV.5. *The runtime of the algorithm in Figure 5 is bounded polynomially in t and n.*

A. Optimizations

There are some straightforward ways in which the above algorithm can be optimized for efficient implementations:

- If $dbf_T(t)$ is to be calculated for several different t (which is the case for the feasibility test), demand triples need to be calculated only once. In that case, the largest of all t is used as a bound for both k and d, and the obtained demand triples can be reused for all t in question.
- The sets DT_k do not need to be kept separated, but can rather be implemented as a common store of all already visited demand triples.

This last idea makes it possible to detect that a demand triple has already been considered some time before (possibly representing a shorter path). More importantly, it also enables a more sophisticated optimization as follows. The key idea is that not only can a demand triple be abandoned if it *itself* has already been considered before, but even if a different, *dominating* one has already been considered. By this we mean that the new demand triple would not contribute new information to the calculation of $dbf_T(t)$, neither by itself nor by any future extension. We illustrate this idea with the following example.

Example IV.6. *In our running example task T from Figure 2, consider again path $\pi = (v_4, v_2, v_3)$ from Example II.2. We know that it corresponds to demand triple $\langle 9, 43, v_3\rangle$ as we calculated before. Another path also ending in v_3 is $\pi' = (v_3, v_1, v_2, v_3)$ and we find that it corresponds to demand triple $\langle 9, 44, v_3\rangle$.*

Intuitively, π generates the same execution demand as π' but during a shorter time interval. Since they both end in the same vertex, they have the same extensions, and thus all paths prefixed with π will always dominate *those prefixed with π'. Clearly, π' is not* critical *for further considerations. Thus, even though the demand triples are not equal, $\langle 9, 44, v_3\rangle$ can be discarded directly. Each demand triple that later on would be based on it would always have a counterpart based on $\langle 9, 43, v_3\rangle$ with a strictly smaller deadline.*

In general, we can define this concept as follows.

Definition IV.7 (Domination)**.** *Let $\xi_1 = \langle e_1, d_1, u\rangle$ and $\xi_2 = \langle e_2, d_2, v\rangle$ be two demand triples. We say that ξ_1* dominates *ξ_2, written $\xi_1 \succcurlyeq \xi_2$, if*

$$e_1 \geqslant e_2,\ d_1 \leqslant d_2,\ \text{and } u = v.$$

We say that ξ_1 strictly dominates *ξ_2, written $\xi_1 \succ \xi_2$, if $\xi_1 \succcurlyeq \xi_2$ and $\xi_1 \neq \xi_2$.*

Definition IV.8 (Critical paths)**.** *A finite path π in $G(T)$ corresponding to a demand triple ξ is* critical*, if there is no path π' corresponding to a demand triple ξ' with $\xi' \succ \xi$. We also say that ξ is critical.*

The proposed optimization is now to discard all demand triples which clearly are not critical because the procedure already has discovered a dominating one. By keeping demand triples per end vertex sorted by deadline, this optimization can be implemented very efficiently, resulting in a highly generalized variant of the dbf procedure presented in [9] for DAGs. In fact, our prototype implementation showed a huge performance improvement using this optimization (from several minutes for large tasks down to a few seconds), since calculating demand pairs for $dbf(t)$ becomes roughly linear in t instead of being quadratic. Further, correctness is guaranteed by the following lemma. It directly implies that demand triples corresponding to non-critical paths can be discarded, since any extension will be non-critical as well. The proof is given in Appendix B.

Lemma IV.9 (Optimal Substructure)**.** *For each critical path $\pi = (\pi_0, \ldots, \pi_l)$, all prefixes $\pi' = (\pi_0, \ldots, \pi_j)$, $j \leqslant l$ are critical as well.*

V. Calculating The Bound

We will now derive a bound for t_f, the supposed counterexample that witnesses the falsity of the condition in Proposition III.4 for infeasible task sets.

With such a bound D, we could run the following feasibility test. Given a task set τ, check $dbf(t) \leqslant t$ for all $t \leqslant D$. If and only if the test succeeds for all $t \leqslant D$, then τ can be scheduled on a preemptive uniprocessor (using EDF). Otherwise, a counterexample t_f is found and τ is shown to not be schedulable. Further, if we can show that D is polynomially bounded in the parameters of the task set, the sketched test is tractable.

Utilization: A central concept we use is the *utilization* of a task set. Intuitively, it describes the maximum execution demand rate that the task set may create *asymptotically*.

Definition V.1 (Utilization)**.** *For a task set τ, a task T and a cycle $\pi = (\pi_0, \ldots, \pi_l)$ in $G(T)$, i.e., $\pi_0 = \pi_l$ and $l \geqslant 1$, we define their* utilizations*:*

$$U(\pi) := \frac{\sum_{j=0}^{l-1} e(\pi_j)}{\sum_{j=0}^{l-1} p(\pi_j, \pi_{j+1})}$$
$$U(T) := \max\{U(\pi) \mid \pi \text{ is a cycle in } G(T)\}$$
$$U(\tau) := \sum_{T \in \tau} U(T).$$

We call the cycle with utilization $U(T)$ the most dense *cycle.*

(We show how to calculate $U(\tau)$ below in Section V-A.)

Cleary, if the utilization of a task set exceeds 1, then it can not be schedulable, since after sufficiently long cycling of each task in its most dense cycle, the system will be overloaded. On the other hand, if the utilization is smaller than 1, we can show that for sufficiently long intervals, the execution demand will be strictly less than the interval size. For an intuitive explanation, see Figure 6. Note that this does *not* mean that $U(\tau) < 1$ is sufficient for feasibility. It only guarantees the existence of a bound D for t_f.

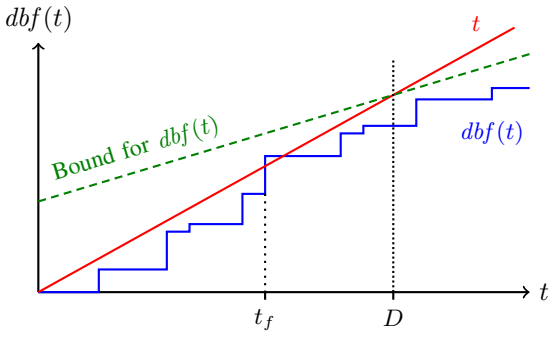

Fig. 6. Demand bound function of some task set. This task set is not schedulable since there is t_f with $dbf(t_f) > t_f$. Let's assume there is a bound for $dbf(t)$ with a slope of less than 1. This bound intersects t at some point D. Clearly, t_f is at most D, so only values up to D need to be checked. For the slope of the bound, we use the task set's utilization.

Formally, we first derive a bound for the demand bound function of each individual task. Each path in $G(T)$ can be partitioned into two types of subpaths: *maximal cycles* and all *remaining* vertices. For all maximal cycles, their execution demand can be bounded by the demand that the most dense cycle could generate in the same amount of time. Further, each vertex of $G(T)$ belongs at most once to the remaining vertices (otherwise, some cycle was not chosen maximal), therefore a bound for the execution demand of the remaining vertices is just the sum of all execution times of all vertices[2].

We express this observation in the following lemma, using $e^{sum}(T) := \sum_{v \in G(T)} e(v)$. For a formal proof, see Appendix C.

Lemma V.2. *For a task T and all t,*

$$dbf_T(t) \leqslant t \cdot U(T) + e^{sum}(T).$$

The above bound for all $dbf_T(t)$ can be used to derive an upper bound for a witness t_f with $dbf(t_f) > t_f$, as follows.

Lemma V.3. *For τ with $U(\tau) < 1$ which is not preemptive uniprocessor feasible, there is a t_f with $dbf(t_f) > t_f$ such that:*

$$t_f < \frac{\sum_{T \in \tau} e^{sum}(T)}{1 - U(\tau)}$$

Proof: Use Lemma V.2 in $t_f < \sum_{T \in \tau} dbf_T(t_f)$ and simple arithmetics. ∎

Thus, only all t up to the specified bound need to be checked. Further, for task sets with $U(\tau) \leqslant c$ for a constant $c < 1$, the bound guarantees that the feasibility problem is tractable by using our proposed algorithm. We state our main technical result in the following theorem.

Theorem V.4. *For a DRT task set τ with $U(\tau) \leqslant c$ for some constant $c < 1$, feasibility can be decided in pseudo-polynomial time.*

Proof: We first note that for $U(\tau) \leqslant c$, we have

$$\frac{\sum_{T \in \tau} e^{sum}(T)}{1 - U(\tau)} \leqslant \frac{\sum_{T \in \tau} e^{sum}(T)}{1 - c}.$$

The term on the right-hand side is clearly polynomial in the values of the task parameters, so it is pseudo-polynomial in the system specification.

Further, given a τ with bounded utilization, we can compute $dbf(t)$ for all t smaller than the bound for t_f by running the algorithm in Figure 5 for each task T. Each time it is called, it returns $dbf_T(t)$ in time polynomial in t and n (Lemma IV.5). From the bound for t_f it follows that the whole procedure is pseudo-polynomial. ∎

Of course, efficient implementations can use an integrated procedure, which calculates all demand triples up to the given bound just once, while checking the condition $dbf(t) \leqslant t$. All optimizations discussed at the end of Section IV apply.

[2]These bounds can of course be tightened, significantly improving analysis performance. However, for simplicity of presentation, we use the presented bound, which is already sufficient for the complexity result.

A. Calculating the Utilization

So far, we only used the utilization of tasks in definitions and proofs, but how can it actually be *computed*? In contrast to simpler task models, this question does not have a straightforward answer, since our notion of a task's utilization is inherently non-trivial. We now present an efficient way of computing $U(T)$ based on an iterative procedure similar to the computation of $dbf_T(t)$ in Section IV.

In general, we need to find the maximum utilization that a cycle π in $G(T)$ can possibly have (Definition V.1). The main problem is that there are potentially infinitely many cycles in $G(T)$, since the definition is not restricted to *simple* cycles (that is, cycles which do not visit any vertex twice, except the last one). The key observation for an efficient construction is that it actually *is* sufficient to restrict the attention to simple cycles. The reason is that any cycle can always be transformed into a simple one with a resulting utilization of no less than the original cycle, as follows. If it contains sub-cycles, then either at least one of them has a higher utilization, or all sub-cycles can be removed. We state this observation formally in the following lemma. A full proof is given in Appendix D.

Lemma V.5. *For each cycle π, there is a* simple *cycle π' with $U(\pi') \geqslant U(\pi)$.*

Example V.6. *The digraph of the example task in Figure 2 contains exactly the following simple cycles (up to rotation):*

$$\pi^D = (v_1, v_2, v_3, v_1), \qquad U\left(\pi^D\right) = 6/36,$$
$$\pi^E = (v_1, v_5, v_4, v_2, v_3, v_1), \qquad U\left(\pi^E\right) = 12/76,$$
$$\pi^F = (v_2, v_4, v_2), \qquad U\left(\pi^F\right) = 6/40.$$

According to the above lemma, no other cycle in $G(T)$ can have a higher utilization. Consequently, we know that the utilization of T is $U(T) = U(\pi^D) = 3/18$.

The computation of $U(T)$ can be based on this observation, since we can restrict the search space for the worst-case cycle to cycles of at most length n. A naïve search via explicit enumeration would still be an exponential procedure. However, we can reuse our path abstraction framework which was already useful to reduce the complexity of the $dbf(t)$ computation. We need to adjust the *demand triple* abstraction idea so that:

1) e now does *not* include the execution demand of the last vertex.
2) d now does *not* include the deadline of the last vertex (but it *does* include the last inter-release seperation). Therefore, we call it p instead.
3) We include the *start vertex*, in addition to the *end vertex*.

Formally, we capture this as follows (see also Figure 7).

Definition V.7 (Utilization Triple)**.** *For a finite path $\pi = (\pi_0, \ldots, \pi_l)$ in $G(T)$, we call $\langle e, p, (\pi_0, \pi_l)\rangle$ a* utilization triple*, if*

$$e = \sum_{i=0}^{l-1} e(\pi_i) \quad \textit{and} \quad p = \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}).$$

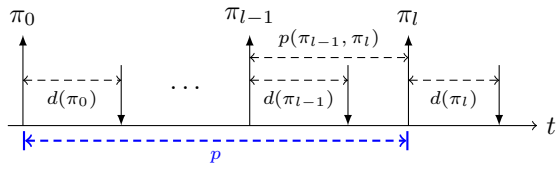

Fig. 7. Illustration of p in a utilization triple $\langle e, p, (\pi_0, \pi_l)\rangle$ corresponding to path $\pi = (\pi_0, \ldots, \pi_l)$. Note that if $\pi_0 = \pi_l$, then π is a cycle and p is the duration of that cycle.

Example V.8. *Consider path $\pi = (v_4, v_2, v_3)$ from the running example in Figure 2. For calculating the utilization, it would be abstracted as the utilization triple $\langle 6, 35, (v_4, v_3)\rangle$.*

Using this abstraction, we are particularly interested in utilization triples $\langle e, p, (u, v)\rangle$ where start and end vertex are the same, $u = v$. In that case, they abstract a cycle and e/p is the utilization of that cycle.

For computing $U(T)$, we can run the algorithm from Figure 5 with the utilization triple abstraction instead. Apart from changing initialization and update procedures of sets UT_k (replacing DT_k) accordingly, we run the outermost loop just for $k = 1, \ldots, n$ and return the following:

$$\max\left\{\tfrac{e}{p} \mid \langle e, p, (u, v)\rangle \in \textstyle\bigcup_{0<k\leqslant n} UT_k \text{ and } u = v\right\}$$

(If the set is empty, i.e., no cycles exist, we return 0.)

Clearly, this procedure generates utilization triples corresponding to all paths of length $k \leqslant n$. These must include all simple paths and therefore all simple cycles. Their utilizations can be directly derived from the utilization triples, and the maximum is returned. Finally, the procedure clearly has pseudo-polynomial runtime complexity. All optimizations discussed at the end of Section IV can be used for efficient implementations of this procedure as well.

VI. Arbitrary Deadlines

We now relax the deadline constraints, allowing $d(u) > p(u, v)$ for the edges (u, v) in $G(T)$. We assume no additional restrictions, i.e., jobs may execute as soon as they are released. In particular, they do not have to wait until preceding jobs are finished.

This extended setting imposes an additional analysis challenge, illustrated by the following example.

Example VI.1. *Consider the DRT task in Figure 8. Since $d(v_2) > p(v_2, v_3) + d(v_3)$, the (absolute) deadline of a job may be* after *the deadline of its succeeding job. An example of this is the job sequence that is sketched in Figure 8 as well.*

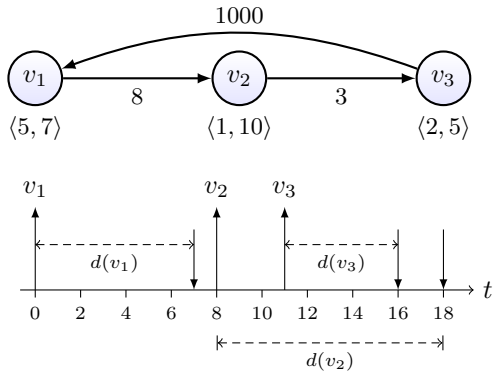

Fig. 8. A DRT task with arbitrary deadlines and a possible job release sequence. Note that (absolute) deadline order does not equal release order.

In this example, path (v_1, v_2, v_3) *corresponds to* two *demand pairs:*

- $\langle 7, 16\rangle$*, because within* 16 *time units, all three jobs are released, but only the first and third are having their deadlines. Together, they have an execution demand of* 7 *time units.*
- $\langle 8, 18\rangle$*, since with additional* 2 *time units, also the second job has its deadline within the considered interval. Thus, all three job execution demands have to be added, resulting in* 8 *time units.*

We see from this example that our procedure from Section IV needs to be extended in order to deal with arbitrary deadlines. We can no longer implicitly assume that the last job of each finite job sequence also has the latest deadline.

For dealing with the general case, we first augment the notion of a path. The extra information we add should represent the subset of its vertices that we consider when calculating execution demand and deadline for a demand pair.

Definition VI.2 (Marked Path)**.** *For a path* $\pi = (\pi_0, \dots, \pi_l)$ *in* $G(T)$*, we call a function* α *a* marking *of its vertices, if* $\alpha(0) = \bullet$ *and* $\alpha(j) \in \{\bullet, \circ\}$ *for all remaining* j*. A path* π *together with its marking* α *is a* marked path $\hat{\pi} := (\pi, \alpha)$.

As shorthand notation, we attach to each vertex its individual marking when writing the path. For example, the path corresponding to the first demand pair $\langle 7, 16\rangle$ from Example VI.1 can be written as $(v_1^\bullet, v_2^\circ, v_3^\bullet)$, since v_2 was not considered for $\langle 7, 16\rangle$ and is thus marked with $\circ$.

In order to determine the corresponding demand pair for a marked path $\hat{\pi}$, we now only consider its $\bullet$-marked vertices[3]. The execution demand $e(\hat{\pi})$ is the sum of their individual execution demands. The deadline $d(\hat{\pi})$ is derived from the *latest* deadline among all $\bullet$-marked vertices. Formally, we define both as follows.

[3]Note that the initial vertex is always $\bullet$-marked, which is a technicality for simplification of presentation without loss of generality.

Definition VI.3. *For a marked path* $\hat{\pi}$*, let* $\alpha^{-1}(\bullet)$ *denote the positions of the* $\bullet$*-marked vertices in* $\hat{\pi}$*. Using this, we define:*

$$e(\hat{\pi}) := \sum_{j \in \alpha^{-1}(\bullet)} e(\pi_j),$$

$$d(\hat{\pi}) := \max_{j \in \alpha^{-1}(\bullet)} \left\{ \sum_{i=0}^{j-1} p(\pi_i, \pi_{i+1}) + d(\pi_j) \right\}.$$

As before, $\langle e(\hat{\pi}), d(\hat{\pi})\rangle$ *denotes a* demand pair.

With these notions, we now show how to extend the procedures from Sections IV and V for calculating both $dbf(t)$ and the bound for t_f.

Calculating Demand Pairs: The additional challenge when calculating demand pairs is the problem of dealing with different markings of the same path. As in Section IV, we would like to apply the iterative generation using demand triples by having each demand triple now correspond to a *marked* path. Consider for example a path $\pi = (\pi_0^\bullet, \pi_1^\circ, \pi_2^\bullet, \pi_3^\circ, \pi_4^\circ)$ which we want to extend with another vertex v. There are two possibilities, $\pi' = (\pi_0^\bullet, \pi_1^\circ, \pi_2^\bullet, \pi_3^\circ, \pi_4^\circ, v^\bullet)$ and $\pi'' = (\pi_0^\bullet, \pi_1^\circ, \pi_2^\bullet, \pi_3^\circ, \pi_4^\circ, v^\circ)$. Thus, given a demand triple ξ for π, we have to generate one for π' and one for π''. However, this can not be done directly from ξ. It includes only a deadline (of either π_0 or π_2, whichever comes later). What is lost is the information about when the job triggered by π_4 is released, but we need this information in order to be able to generate the demand triple for π'.

Our solution for this is to extend the abstraction to *demand quadruples*.

Definition VI.4 (Demand Quadruple)**.** *For a marked path* $\hat{\pi} = (\pi_0, \dots, \pi_l)$ *in* $G(T)$*, we define its* duration *as*

$$p(\hat{\pi}) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}).$$

With $e(\hat{\pi})$ *and* $d(\hat{\pi})$ *from Definition VI.3, we call* $\langle e(\hat{\pi}), d(\hat{\pi}), p(\hat{\pi}), \pi_l\rangle$ *a* demand quadruple.

Clearly, there are only pseudo-polynomially many demand quadruples with $e, d, p \leqslant D$ for some bound D. Thus, we can use the iterative algorithm from Figure 5 in Section IV with the demand quadruple abstraction in order to obtain a pseudo-polynomial procedure for calculating $dbf_T(t)$. For each $\langle e, d, p, u\rangle$ chosen in line 4 and edge (u, v) from line 5, we create *two* new demand quadruples: $\langle e', d', p', v\rangle$ for extension with a $\bullet$-marked v and $\langle e'', d'', p'', v\rangle$ for a $\circ$-marked v. Their calculation is straightforward:

$$\begin{aligned} e' &= e + e(v) & e'' &= e \\ d' &= \max(d, p + p(u, v) + d(v)) & d'' &= d \\ p' &= p + p(u, v) & p'' &= p + p(u, v) \end{aligned}$$

It can be easily shown that with these changes, the procedure is still correct. Further, with a suitable domination relation $\succ$, all optimizations discussed in Section IV-A can be applied for efficient implementations.

Example VI.5. *Consider the DRT task from Figure 8. Vertex v_1 can be interpreted as a marked 0-path $(v_1^\bullet)$ and thus corresponds to demand quadruple $\langle 5, 7, 0, v_1\rangle$. We can extend it with v_2, which gives the possibilities $\langle 6, 18, 8, v_2\rangle$ and $\langle 5, 7, 8, v_2\rangle$, corresponding to marked paths $(v_1^\bullet, v_2^\bullet)$ and $(v_1^\bullet, v_2^\circ)$, respectively. Note that if we extend both of these with a $\bullet$-marked v_3, we get the demand quadruples $\langle 8, 18, 11, v_3\rangle$ and $\langle 7, 16, 11, v_3\rangle$ from which both demand pairs in Example VI.1 are derived.*

Calculating the Bound: The bound for t_f that we derived in Section V turns out to hold also for the case of arbitrary deadlines. A first observation is that the bound in Lemma V.2 does not depend on any deadlines.

More formally, given a task T, we transform it into a task T' where we reduce the deadlines such that T' now satisfies the Frame Separation property[4]. Clearly, $dbf_T(t) \leqslant dbf_{T'}(t)$ for all t. Further, we know that $U(T) = U(T')$ and $e^{sum}(T) = e^{sum}(T')$ since both their definitions are independent from deadlines. It follows directly that Lemma V.2 holds even for arbitrary-deadline task sets.

Finally, this implies that also Lemma V.3 must hold. We summarize all insights from this section in the following theorem.

Theorem VI.6. *For a DRT task set τ with arbitrary deadlines and $U(\tau) \leqslant c$ for some constant $c < 1$, feasibility can be decided in pseudo-polynomial time.*

VII. Conclusions and Future Work

In this paper we have presented a new task model that allows high expressiveness when modeling real-time systems. It overcomes restrictions of previous models that prevent appropriate modeling of many systems. In order to approach the feasibility problem, we have introduced a path abstraction framework that allows efficient calculation of not only a task set's execution demand, but also of its utilization. Our technique further applies to the analysis of arbitrary deadline systems, and allows non-trivial optimizations to significantly increase analysis efficiency in implementations. We have shown that our proposed method has pseudo-polynomial complexity for task systems with bounded utilization, which is considered *tractable* by the real-time scheduling community. By presenting a significant generalization of the non-cyclic RRT (previously being the most general of all tractable models), our model is therefore the most general tractable model currently known.

As future work, we seek to evaluate our techniques on real-world case studies using more optimized implementations. Further, we wish to determine where the feasibility problem of models becomes strongly NP- or $coNP$-hard. Beyond this border, it provably cannot be solved both efficiently *and* precisely anymore (assuming $P \neq NP$). Our ongoing work indicates that the DRT model already is very close.

[4]This may lead to $e(v) > d(v)$ for some vertices v in T', but does not cause any technical difficulty, since we do *not* demand equivalence of T and T' in terms of schedulability.

References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Inf. Comput.*, vol. 205, no. 8, pp. 1149–1172, 2007.

[3] S. K. Baruah, "The non-cyclic recurring real-time task model," in *RTSS 2010*, to appear.

[4] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Cambridge, MA, USA, Tech. Rep., 1983.

[5] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," *IEEE Trans. Softw. Eng.*, vol. 23, no. 10, pp. 635–645, 1997.

[6] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Syst.*, vol. 17, no. 1, pp. 5–22, 1999.

[7] S. K. Baruah, "Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks," *Real-Time Syst.*, vol. 24, no. 1, pp. 93–128, 2003.

[8] N. Tchidjo Moyo, E. Nicollet, F. Lafaye, and C. Moy, "On schedulability analysis of non-cyclic generalized multiframe tasks," in *ECRTS 2010*, pp. 271–278.

[9] S. Chakraborty, T. Erlebach, and L. Thiele, "On the complexity of scheduling conditional real-time code," in *WADS 2001*, pp. 38–49.

[10] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS 2000*, vol. 4.

Appendix

A. Proof of Lemma IV.4

Proof: For the first part, let e be the output of our algorithm in Figure 5 given input t. We want to show that there is a path π in $G(T)$ with $e(\pi) = e$ and $d(\pi) \leqslant t$. From the return statement (line 14) it follows that there must have been a demand triple $\langle e, d, v\rangle \in DT_k$ for some $k \leqslant t$ with $d \leqslant t$. It is thus sufficient to show that for each demand triple in each DT_k, there is a corresponding path of length k in $G(T)$. This can be shown by induction on k:

- $k = 0$: All demand triples in DT_0 are generated in line 1 directly from the vertices of $G(T)$ which are 0-paths.
- $k - 1 \rightsquigarrow k$: Given $\langle e', d', v\rangle \in DT_k$, it must have been added to DT_k in line 9. The demand triple $\langle e, d, u\rangle$ from which it was created must have been chosen in line 4 from DT_{k-1}. By induction hypothesis, there is a path $\pi = (\pi_0, \ldots, \pi_{k-2}, u)$ of length $k - 1$ in $G(T)$ corresponding to $\langle e, d, u\rangle$. Further, there is an edge (u, v) in $G(T)$ that has been chosen in line 5. Thus, $\pi' = (\pi_0, \ldots, \pi_{k-2}, u, v)$ is a path of length k in $G(T)$. Finally, the calculations in lines 6 and 7 ensure that π' corresponds to $\langle e', d', v\rangle$.

For the second part, it is sufficient to show that all paths π of length k in $G(T)$ with a deadline of at most t have a corresponding demand triple $\langle e, d, v\rangle$ in DT_k. Again, we show this by induction on k.

- $k = 0$: For all 0-paths, i.e., all vertices, all demand triples are included in DT_0 via line 1.
- $k - 1 \rightsquigarrow k$: Given a path $\pi = (\pi_0, \ldots, \pi_{k-2}, u, v)$ of length k with deadline $d(\pi) \leqslant t$, we know by induction hypothesis that $\pi' = (\pi_0, \ldots, \pi_{k-2}, u)$ of length $k - 1$ has a corresponding demand triple $\langle e, d, u\rangle$ in DT_{k-1}. This demand triple is eventually chosen in line 4 and also the edge (u, v) in line 5. Finally, lines 6 and 7 calculate the corresponding demand triple for π which will be added to DT_k in line 9 since $d' = d(\pi) \leqslant t$ by assumption. ∎

B. Proof of Lemma IV.9

Proof: Intuitively, if some prefix π^{pfx} of a critical path π is not critical, then π^{pfx} is dominated by another path π'. This π' could replace π^{pfx} in π, resulting in a path that dominates π (and thus contradicts that π is critical).

Formally, we show this by induction on the path length l.

$l = 0$: A 0-path has no prefixes except itself.

$l - 1 \rightsquigarrow l$: Given a critical path $\pi = (\pi_0, \ldots, \pi_l)$, assume that its $(l-1)$-prefix $\pi^{pfx} = (\pi_0, \ldots, \pi_{l-1})$ is not critical. Then there must be a $\pi' = (\pi'_0, \ldots, \pi'_k)$ such that

$$\langle e(\pi'), d(\pi'), \pi'_k \rangle \succ \left\langle e(\pi^{pfx}), d(\pi^{pfx}), \pi_{l-1} \right\rangle.$$

In particular, $\pi'_k = \pi_{l-1}$. We can now use π' to replace the prefix π^{pfx} of π. Consider the new path

$$\pi'' = (\underbrace{\pi'_0, \ldots, \pi'_k}_{\pi'}, \pi_l).$$

We have that

$$\begin{aligned} e(\pi'') &= e(\pi') + e(\pi_l) \geqslant e(\pi^{pfx}) + e(\pi_l) = e(\pi), \\ d(\pi'') &= d(\pi') - d(\pi'_k) + p(\pi'_k, \pi_l) + d(\pi_l) \\ &\leqslant d(\pi^{pfx}) - d(\pi'_k) + p(\pi'_k, \pi_l) + d(\pi_l) \\ &= d(\pi^{pfx}) - d(\pi_{l-1}) + p(\pi_{l-1}, \pi_l) + d(\pi_l) \\ &= d(\pi). \end{aligned}$$

Consequently, the new path π'' dominates π, i.e.,

$$\langle e(\pi''), d(\pi''), \pi_l \rangle \succ \langle e(\pi), d(\pi), \pi_l \rangle.$$

This contradicts that π is critical. Thus, π^{pfx} must be critical as well. By induction hypothesis, all prefixes of π^{pfx} are also critical, which completes the proof for all prefixes of π. ∎

C. Proof of Lemma V.2

Proof: For a given t, let π be a path with maximal execution demand during any interval of length t, i.e., $e(\pi) = dbf_T(t)$ and $d(\pi) \leqslant t$. If π is *simple*, we have $e(\pi) \leqslant e^{sum}(T)$ and are done. Otherwise, we first identify all *cycles* in π as follows. For each vertex v in $G(T)$, let $m(v)$ denote the number of occurrences of v in π. Find the first vertex π_i in π with $m(\pi_i) > 1$. Let π_j be the last occurrence of π_i in π. We have $i < j$ and call $\pi^{(1)} = (\pi_i, \ldots, \pi_j)$ the *first cycle* in π. We remove it from π by considering the new path

$$\pi' = (\pi_0, \ldots, \pi_i, \pi_{j+1}, \ldots, \pi_l).$$

Note that we cut away the *edges* of $\pi^{(1)}$ from π which only leaves the start vertex π_i (that equals π_j) of $\pi^{(1)}$ in π.

We repeat this procedure until no new cycles are found. Let $\pi^{(j)}$ denote the j-th cycle found by this procedure and π^* the resulting path (π with all cycles removed). Clearly, π^* is simple. Note that the $\pi^{(j)}$ do *not* need to be simple, i.e., they may contain sub-cycles.

For each $\pi^{(j)}$, let l_j denote its length. By construction, we know now (via reordering) that

$$e(\pi) = e(\pi^*) + \sum_j \sum_{i=0}^{l_j - 1} e\left(\pi_i^{(j)}\right). \tag{1}$$

Clearly, $e(\pi^*) \leqslant e^{sum}(T)$ since π^* is simple. Further, for all cycles $\pi^{(j)}$ we have

$$\begin{aligned} \sum_{i=0}^{l_j - 1} e\left(\pi_i^{(j)}\right) &= U\left(\pi^{(j)}\right) \cdot \sum_{i=0}^{l_j - 1} p\left(\pi_i^{(j)}, \pi_{i+1}^{(j)}\right) \\ &\leqslant U(T) \cdot \sum_{i=0}^{l_j - 1} p\left(\pi_i^{(j)}, \pi_{i+1}^{(j)}\right). \end{aligned}$$

This holds because $U(T)$ is the *maximal* utilization over *all* cycles. Summing over all cycles of π, we get

$$\begin{aligned} \sum_j \sum_{i=0}^{l_j - 1} e\left(\pi_i^{(j)}\right) &\leqslant U(T) \cdot \underbrace{\sum_j \sum_{i=0}^{l_j - 1} p\left(\pi_i^{(j)}, \pi_{i+1}^{(j)}\right)}_{\leqslant d(\pi) \leqslant t} \\ &\leqslant U(T) \cdot t. \end{aligned}$$

Finally, we apply this to Equation (1) and we get

$$dbf_T(t) = e(\pi) \leqslant t \cdot U(T) + e^{sum}(T).$$

∎

D. Proof of Lemma V.5

Proof: Given a cycle $\pi = (\pi_0, \ldots, \pi_l)$, we show the existence of a *simple* cycle π' with $U(\pi') \geqslant U(\pi)$ by induction on l. Clearly, a cycle of length $l = 1$ is simple, so the base case is trivial. For the induction step, we first identify the sub-cycles $\pi^{(j)}$ as in the proof of Lemma V.2. (To be precise, this time $m(v)$ does not consider the end vertex of π.) If there are no sub-cycles, we are done, since π must be simple in that case. Otherwise, there are two cases for the utilizations of the sub-cycles:

First case: $\exists j : U\left(\pi^{(j)}\right) \geqslant U(\pi)$. In this case, since $\pi^{(j)}$ is shorter than π, we know by induction hypothesis that there is a simple cycle π' with $U(\pi') \geqslant U\left(\pi^{(j)}\right)$ and we are done.

Second case: $\forall j : U\left(\pi^{(j)}\right) < U(\pi)$. We now know that all sub-cycles of π have a lower utilization than π itself. Thus, we construct π' by removing all sub-cycles, i.e., π' is the π^* from the procedure of identifying the sub-cycles. For all j, let l_j denote the length of $\pi^{(j)}$ as before, and i_j denote the position of the first vertex of $\pi^{(j)}$ in π. With this we can write

$$\pi' = (\pi_0, \ldots, \pi_{i_1}, \pi_{i_1 + l_1 + 1}, \ldots, \pi_l).$$

For its utilization, we get

$$\begin{aligned} U(\pi') &= \frac{\sum_{i=0}^{l-1} e(\pi_i) - \sum_j \sum_{i=i_j}^{i_j + l_j - 1} e(\pi_i)}{\sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) - \sum_j \sum_{i=i_j}^{i_j + l_j - 1} p(\pi_i, \pi_{i+1})} \\ &\geqslant \frac{\sum_{i=0}^{l-1} e(\pi_i)}{\sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1})} = U(\pi). \end{aligned}$$

Note that the last inequality holds since for all $a, b, c_i, d_i \in \mathbb{N}_{>0}$ with $a > \sum_i c_i > 0$ and $b > \sum d_i > 0$:

$$\left[\forall i : \frac{a}{b} \geqslant \frac{c_i}{d_i}\right] \implies \frac{a - \sum_i c_i}{b - \sum_i d_i} \geqslant \frac{a}{b}$$

∎